Fundamentals Of Computer Science

Course Notes

Felipe Balbi

June 17, 2020

Contents

Week 1 **5**
1.101 Introduction to propositional logic 5
1.103 Building blocks of logic 5
1.105 Truth Table: examples 7
1.202 Tautology and consistency 10

Week 2 **12**
1.301 Equivalences 12
1.304 First-order logic 13

Week 3 **16**
2.01 What is a proof? 16
2.101 Direct proof 16
2.102 Direct proof examples 18
2.202 Proof by contradiction 19
2.203 Proof by contraposive 20

Week 4 **22**
2.301 Proof by induction 22
2.303 Example of a correct proof 23
2.305 Example of an incorrect proof 24
2.401 Conclusion 25

Week 5 **26**
3.01 Introduction 26
3.101 Counting 26
3.102 Complex counting 27
3.201 The Pigeonhole Principle 28
3.202 The Pigeonhole Principle: examples 29

Week 6 **31**
3.301 Permutations 31
3.304 Combinations 32

Week 7 **35**
4.01 Introduction 35
4.101 Basic definitions, letters, strings 35

4.103 What is an automaton? . . . 36
4.201 Finite automata: example . . . 38
4.202 Language of the automata . . . 39
4.204 Recognise a language . . . 43

Week 8 **45**
4.301 Deterministic finite automata (DFA) vs nondeterministic finite automata (NFA) . . . 45
4.303 Computation by NFA . . . 47

Week 9 **49**
5.101 Regular expressions . . . 49
5.103 Design regular expressions . . . 52
5.201 Regular expressions and finite automata . . . 53
5.203 Regular expressions and finite automata: examples . . . 54

Week 10 **57**
5.301 Regular or non-regular? . . . 57
5.303 Pumping lemma . . . 59

Week 11 **60**
6.01 Introduction . . . 60
6.101 Grammar . . . 60
6.201 Language of a grammar . . . 62
6.203 Designing a grammar . . . 63

Week 12 **65**
6.301 Regular expression to context-free grammar . . . 65
6.304 Chomsky Normal Form . . . 67

Week 13 **71**
7.101 Turing machines . . . 71

Week 14 **74**
7.301 The power of Turing machines . . . 74
7.303 The language of Turing machines . . . 75

Week 15 **77**
8.101 What is an algorithm ? . . . 77
8.103 Representing algorithms . . . 78
8.201 Simple algorithms: insertion sort . . . 79
8.203 Simple algorithms: bubble sort . . . 80

Week 16 **81**
8.301 Binary search . . . 81

8.304 Heap sort 82

Week 17 **91**
9.101 Recursion 91
9.103 Quick sort 93
9.201 Merging lists 94
9.203 Merge sort 101

Week 18 **102**
9.301 The Algorithm of Happiness 102
9.303 The Gale-Shapley algorithm: example and pseudocode 104

Week 19 **106**
10.101 Efficiency: insertion sort (time complexity) 106
10.103 Efficiency: bubble sort and binary search 106
10.201 Asymptotic complexity 107
10.203 Big O notation 107

Week 20 **109**
10.301 Recursion complexity 109
10.304 Efficiency: Quick sort, merge sort 110

Week 1

Learning Objectives:

- Understand logical arguments and apply basic concepts of formal proof.

Essential Reading

- Rosen, K.H. Discrete mathematics and its applications (New York: McGraw-Hill, 2012) 7th edition, Chapter 1.1-1.2, pp.1-22.

1.101 Introduction to propositional logic

Propositional Logic is a system that deals with propositions or statements.

Below there's an example of where we can apply propositional logic to derive conclusions.

Liars And Knights

Imagine there is an island with two types of people. Liars who always tell lies and knights who always tell the truth. One an excursion, you visit the island and encounter two people, person A and person B. Person A says "at least one of us is a liar", while person B says nothing. What conclusion can you draw?

With a little logical thinking, we can conclude that person A is a knight and person B is a liar.

1.103 Building blocks of logic

What is a proposition?

A **proposition** is a statement that can be either **true** or **false**. It must be one or the other, never both nor neither.

Examples of proposition:

- 2 is a prime number (T)
- 5 is an even number (F)

Not a proposition:

- x is a prime number

  In this case, it can be made into a proposition by assigning a value to x.

- Are you going to school?

  Because this is a question, we can't assign a truth value to the sentence.

- Do your homework now

  Being an order, it has no truth value.

Syntaxes of the propositional logic

Propositions are denoted by capitable letters: P, Q, R, and so on.

- P = carrots are orange
- Q = I went to a party yesterday

General statements are denoted by lowercase letters: p, q, r, and so on. They carry on a logical argument, are used in proofs, called propositional variables.

Connectives: change or combine propositions

Connectives transform **atomic** propositions into **compound** propositions.

1. Logical NOT ($\neg$)

   $\neg p$ is true if and only if p is false. Also called **negation**.

2. Logical OR ($\vee$)

   $p \vee q$ is true if and only if at least one of p or q is true or if both p and q are true. Also called **disjunction**.

3. Logical AND ($\wedge$)

   $p \wedge q$ is true if and only if both p and q are true and false otherwise. Also called **conjunction**.

4. Logical if then ($\rightarrow$)

   $p \rightarrow q$ is true if and only if either p is false or q is true. Also called **implication** or **conditional**. p is called the **premise** and q is the **conclusion**.

5. Logical if and only if ($\leftrightarrow$)

   $p \leftrightarrow q$ is true if and only if both p and q are true or both are false. Also called **bi-conditional**.

6. Exclusive or ($\oplus$)

   $p \oplus q$ is true if and only if p or q is true but not both.

Translation from Logical Proposition to English

Let:

$$P = \text{I study 20 hours a week}$$
$$Q = \text{I attend all the lectures}$$
$$R = \text{I will pass the exam}$$
$$S = \text{I will be happy}$$

Translate the following statement to English:

- $(P \vee Q) \rightarrow (R \wedge S)$

  If **I study 20 hours a week** or **I attend all the lectures** then **I will pass the exam** and **I will be happy**.

Translation from English to Logical Proposition

Given the statement:

If UK does not exit the EU then skilled nurses will not leave the NHS and research grants will remain intact.

Translating to logical proposition we get:

$$P = \text{UK exits the EU}$$
$$Q = \text{Skilled nurses will leave the NHS}$$
$$R = \text{Research grants will remain intact}$$
$$\neg P \rightarrow \neg Q \wedge R$$

Note that we removed any **connectives** from our propositions as that's a good practice. This makes the logical statement easier to follow.

1.105 Truth Table: examples

A truth table is a set of all outcomes of propositions and connectives. The number of rows in a truth table, depends on the number of given propositions. If we have n propositions, our truth table will have 2^n rows.

Truth tables for each connective

What follows is a list of truth tables for each connective

1. Negation ($\neg$)

p	$\neg p$
1	0
0	1

2. Conjunction ($\wedge$)

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

3. Disjunction ($\vee$)

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

4. Implication ($\rightarrow$)

p	q	$p \rightarrow q$
1	1	1
1	0	0
0	1	1
0	0	1

5. Bi-conditional ($\leftrightarrow$)

p	q	$p \leftrightarrow q$
1	1	1
1	0	0
0	1	0
0	0	1

6. Exclusive Or ($\oplus$)

p	q	$p \oplus q$
1	1	0
1	0	1
0	1	1
0	0	0

Operator Precedence

When formulae are written without parenthesis, we must rely on rules of operator precedence. Logic operator precende rules are as follows:

$\neg \wedge \vee \rightarrow \leftrightarrow$

1. Example

   If we have the logical statement $p \rightarrow p \wedge \neg q \vee s$, we can parse it following the steps below:

$$
\begin{aligned}
&p \rightarrow p \wedge \neg q \vee s \\
&p \rightarrow p \wedge (\neg q) \vee s \\
&p \rightarrow (p \wedge (\neg q)) \vee s \\
&p \rightarrow ((p \wedge (\neg q)) \vee s) \\
&(p \rightarrow ((p \wedge (\neg q)) \vee s))
\end{aligned}
$$

Constructing Truth Tables for Complex Formulae

1. Example 1

p	q	$p \wedge q$	$(p \wedge q) \rightarrow p$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

2. Example 2

p	q	$q \rightarrow p$	$p \wedge (q \rightarrow p)$
1	1	1	1
1	0	1	1
0	1	0	0
0	0	1	0

3. Comparing both examples

p	q	$(p \wedge q) \rightarrow p$	$p \wedge (q \rightarrow p)$
1	1	1	1
1	0	1	1
0	1	1	0
0	0	1	0

1.202 Tautology and consistency

Tautology

A formula that is **always** true regardless of the truth value of the proposition.

p	$\neg p$	$p \vee \neg p$
0	1	1
1	0	1

Consistent

A formula that is true **at least** for one scenario. All connectives are consistent.

The formula $p \wedge \neg p$ is **inconsistent** because it can never be true. Inconsistent formulae are also called **contradictions**.

1.204 Tautology and consistency: examples

1. Example 1: $p \vee (q \wedge \neg r)$

p	q	r	$\neg r$	$q \wedge \neg r$	$p \vee (q \wedge \neg r)$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	1	0	1	1	1
1	1	1	0	0	1

   This is a **Consistent** formula

2. Example 2: $(p \rightarrow q) \rightarrow (\neg q \vee r)$

p	q	r	$p \rightarrow q$	$\neg q$	($\neg$ q $\vee$ \$)	$(p \rightarrow q) \rightarrow (\neg q \vee r)$
0	0	0	1	1	1	1
0	0	1	1	1	1	1
0	1	0	1	0	0	0
0	1	1	1	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	0	0	0
1	1	1	1	0	1	1

   This is a **Consistent** formula

3. Example 3: $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$

p	q	$p \to q$	$\neg p$	$(\neg p \vee q)$	$(p \to q) \leftrightarrow (\neg p \vee q)$
0	0	1	1	1	1
0	1	1	1	1	1
1	0	0	0	0	1
1	1	1	0	1	1

This formula is a **Tautology**

Week 2

Learning Objectives:

- Understand logical arguments and apply basic concepts of formal proof.

Essential Reading

- Rosen, K.H. Discrete mathematics and its applications (New York: McGraw-Hill, 2012) 7th edition, Chapter 1.3–1.4, pp.22–52.

1.301 Equivalences

Formulae A and B are equivalent if they have identical truth tables. Equivalence is denoted by the symbol $\equiv$

In other words, $A \equiv B$ means that A and B have the same truth values, regardless of how variables are assigned.

One thing to note is that $\equiv$ is **NOT** a connective.

De Morgan's Laws

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

Truth Tables

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

p	q	$\neg p$	$\neg q$	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

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

p	q	$\neg p$	$\neg q$	$p \vee q$	$\neg(p \vee q)$	$\neg p \wedge \neg q$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

3. $(p \to q) \equiv (\neg p \vee q) \equiv \neg(p \wedge \neg q)$

p	q	$\neg p$	$\neg q$	$p \to q$	$\neg p \vee q$	$\neg(p \wedge \neg q)$
0	0	1	1	1	1	1
0	1	1	0	1	1	1
1	0	0	1	0	0	0
1	1	0	0	1	1	1

4. Contrapositive: $(p \to q) \equiv (\neg q \to \neg p)$

p	q	$\neg p$	$\neg q$	$p \to q$	$\neg q \to \neg p$
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	0	0
1	1	0	0	1	1

1.304 First-order logic

Important Notions

- **Predicates** describe properties of objects

  A simple example could be `odd(3)`. Here we're applying the predicate `odd` to the object `3`. When arguments are applied to predicates, they become propositions and connectives for propositional logic can be employed in the usual manner:

  $$Odd(3) \wedge Prime(3) = T$$

- **Quantifiers** allow reasoning on multiple objects

  The objects from a quantified statement are chosen from a *Domain*.

  - **Existential Quantifier** $\exists$

    We use it as follows: $\exists x$ some formula.

    When proving a formula based on the existential quantifier, it is enough to find **one** element which makes the formula true. In other words, existentially quantified statements are **false** unless there is a positive example.

  - **Universal Quantifier** $\forall$

    We use it as follows: $\forall x$ some formula.

In order to satisfy the formula, we must prove that **every** x satisfies the formula.

Note that a single counterexample is enough to disprove a universally quantified statement. In other words, universally quantified statements are **true** unless there is a false example.

Translations English - Logic

"All P's are Q's" translates into $\forall x(P(x) \to Q(x))$

"No P's are Q's" translates into $\forall x(P(x) \to \neg Q(x))$

"Some P's are Q's" translates into $\exists x(P(x) \wedge Q(x))$

"Some P's are not Q's" translates into $\exists x(P(x) \wedge \neg Q(x))$

Quantifiers to connectives

- Existential Quantifier

$\exists x P(x)$ and domain $D = \{x_1, x_2, \ldots, x_n\}$. This is equivalent to saying $P(x_1) \vee P(x_2) \vee \ldots \vee P(x_n)$

- Universal Quantifier

$\forall x P(x)$ and domain $D = \{x_1, x_2, \ldots, x_n\}$. This is equivalent to saying $P(x_1) \wedge P(x_2) \wedge \ldots \wedge P(x_n)$

Negation of Quantifiers

- Existential Quantifier

$$\begin{aligned} \neg\exists x P(x) &\equiv \neg(P(x_1) \vee P(x_2) \vee \ldots \vee P(x_n)) \\ &\equiv \neg P(x_1) \wedge \neg P(x_2) \wedge \ldots \wedge \neg P(x_n)) \\ &\equiv \forall x \neg P(x) \end{aligned}$$

- Universal Quantifier

$$\begin{aligned} \neg\forall x P(x) &\equiv \neg(P(x_1) \wedge P(x_2) \wedge \ldots \wedge P(x_n)) \\ &\equiv \neg P(x_1) \vee \neg P(x_2) \vee \ldots \vee \neg P(x_n)) \\ &\equiv \exists x \neg P(x) \end{aligned}$$

1. Example

$$\begin{aligned}\neg(\forall x(P(x) \to Q(x)) &\equiv \exists x \neg(P(x) \to Q(x)) \\ &\equiv \exists x \neg(\neg P(x) \vee Q(x)) \\ &\equiv \exists x(\neg\neg P(x) \wedge \neg Q(x)) \\ &\equiv \exists x(P(x) \wedge \neg Q(x))\end{aligned}$$

Week 3

Learning Objectives:

- Correctly follow a sequence of justified steps to reach a conclusion statement.
- Prove a conclusion statement by first assuming it is false.
- Describe inductive steps.
- Understand logical arguments and apply basic concepts of formal proof.

2.01 What is a proof?

A proof is a sequence of logical statements that explains why a statement is true. Rosen's book defines a proof as follows:

> A proof is a valid argument that establishes the truth of a mathematical statement.

We need proofs to establish general truths about conjectures. For example a computer cannot confirm that **all** numbers have a certain property. Well, considering that numbers are infinite and computers work with finite amounts of memory, a computer will not be able to answer that question.

Given a theorem, often there are many ways in which we can prove it. There are commonly used proof techniques which we learn about.

2.101 Direct proof

A direct proof exploits definitions and other mathematical theorems. It arrives at the desired statement by employing valid logical steps.

A direct proof is:

- Easy because there is no particular technique is used
- Not easy because the starting point is not obvious
- Know your definitions
- Allowed to use any theorem, axiom, logic, etc

Example 1:

Theorem 1. *If n and m are even numbers, then $n + m$ is also even*

Proof. What does even mean?

If an integer is even, it is twice another integer.

$$\begin{aligned} n &= 2k \\ m &= 2l \end{aligned}$$

k and l are integers

$$\begin{aligned} n + m &= 2k + 2l \\ &= 2(k + l) \end{aligned}$$

Let $k + l = t$,

$$n + m = 2t$$

$\therefore n + m$ is even. □

Example 2:

Theorem 2. *$\forall n \in \mathbb{N}, n^2 + n$ is even.*

Proof. If n is even, then $n = 2k$.

$$n^2 + n = (2k)^2 + 2k \,\text{Even}$$

If n is odd, then $n = 2k + 1$.

$$\begin{aligned} n^2 + n &= (2k + 1)^2 + 2k + 1 \\ &= (2k)^2 + 2 \cdot 2k + 1^2 + 2k + 1^2 \\ &= 4k^2 + 6k + 2 \,\text{Even} \end{aligned}$$

□

2.102 Direct proof examples

Example 1:

Theorem 3. *If $a < b < 0$, then $a^2 > b^2$*

Proof. Assume $a < b$ and $a < 0$. Multiplying both sides of the inequality by a gives:

$$\begin{aligned} a \cdot a &< b \cdot a \\ a^2 &> b \cdot a \end{aligned}$$

Assume $a < b$ and $b < 0$. Multiplying both sides of the inequality by b gives:

$$\begin{aligned} a \cdot b &< b \cdot b \\ a \cdot b &> b^2 \end{aligned}$$

By the commutative property of multiplication we know that $a \cdot b = b \cdot a$, therefore:

$$\begin{aligned} a^2 > a \cdot b > b^2 \\ \therefore a^2 > b^2 \end{aligned}$$

□

Example 2:

Theorem 4. *$\forall x \in \mathbb{N}, 2x^3 + x$ is a multiple of 3.*

Proof. Factorizing $2x^3 + x$ gives $x(2x^2 + 1)$.

If x is a multiple of 3, the proof is complete.

If $x = 3k + 1$, then:

$$\begin{aligned} x(2x^2 + 1) &= (3k + 1)[2(3k + 1)^2 + 1] \\ &= (3k + 1)[2(9k^2 + 6k + 1) + 1] \\ &= (3k + 1)(18k^2 + 12k + 3] \\ &= 3(3k + 1)((6k^2 + 4k + 1) \end{aligned}$$

If $x = 3k + 2$, then:

$$\begin{aligned} x(2x^2+1) &= (3k+2)[2(3k+2)^2+1] \\ &= (3k+2)[2(9k^2+12k+4)+1] \\ &= (3k+2)(18k^2+24k+9) \\ &= 3(3k+2)(6k^2+8k+3) \end{aligned}$$

□

2.202 Proof by contradiction

Proof by contradiction is also referred to as *indirect proof*. It follows a simple structure to prove that statement **A** is *true*.

We start by assuming **A** to be *false*, we follow just like a direct proof by employing mathematical definitions, theorems, axioms and logical steps until we arrive at a statement which **contradicts** our original assumption. This would show our original assumption to be incorrect. Therefore, if our assumption is **not** false, then it can only be true.

Example 1:

Theorem 5. *The square-root of two,* $\sqrt{(}2)$ *is irrational*

Proof. Assume $\sqrt{(}2)$ is rational. This means it can be written as a fraction, in lowest terms, of the form $\frac{p}{q}$ for $p, q \in \mathbb{N}$, $q \neq 0$.

If $\frac{p}{q}$ is in lowest terms, it means the fraction cannot be further simplified. Therefore, we have:

$$\begin{aligned} \sqrt{(}2) &= \frac{p}{q} \\ \left(\sqrt{(}2)\right)^2 &= \left(\frac{p}{q}\right)^2 \\ 2 &= \frac{p^2}{q^2} \\ 2q^2 &= p^2 \end{aligned}$$

From this, we can see that p must be even. Which means $p = 2k$. Therefore:

$$\begin{aligned} 2q^2 &= p^2 \\ 2q^2 &= (2k)^2 \\ 2q^2 &= 4k^2 \\ q^2 &= 2k^2 \end{aligned}$$

From this, we can see that q must also be even. Which means our fraction $\frac{p}{q}$ cannot be in lowest terms. Therefore, $\sqrt{(2)}$ cannot be a rational number, so it is irrational. □

Example 2:

Theorem 6. *There is an infinite number of prime numbers.*

Proof. Assume there are finitely many prime numbers. Let the set of prime numbers be $P = \{p_1, p_2, \ldots, p_n\}$. Let $N = (p_1 \cdot p_2 \cdot \ldots \cdot p_n) + 1$.

If we divide N by any of the prime numbers in our list of prime numbers, it will have a remainder of 1. Therefore, N is, itself, a prime number. □

2.203 Proof by contraposive

This technique exploits equivalent classes of logical statements. Let us remember that $a \to b \equiv \neg b \to \neg a$.

In some cases, when we need to prove $a \to b$, it may be easier to prove its contrapositive $(\neg b \to \neg a)$ is true.

Example 1:

Theorem 7. $\forall n \in \mathbb{N}, Odd(n^3 + 1) \to Even(n)$

Proof. By means of the contrapositive $\forall n \in \mathbb{N}, Odd(n) \to Even(n^3 + 1)$.

$$n = 2k + 1 \forall k \in \mathbb{N}$$

$$\begin{aligned} n^3 + 1 &= (2k+1)^3 + 1 \\ &= (2k+1)(2k+1)(2k+1) + 1 \\ &= 8k^3 + 12k^2 + 6k + 2 \\ &= 2(4k^3 + 6k^2 + 3k + 1) \end{aligned}$$

□

Example 2:

Theorem 8. *Suppose* $x, y \in \mathbb{R}$, $y^3 + yx^2 \leq x^3 + xy^2 \to y \leq x$

Proof. By contrapositive $y > x \to y^3 + yx^2 > x^3 + xy^2$.

Assuming $y > x$, we know that $y - x > 0$. Let us multiply both sides of the inequality by $x^2 + y^2$. Therefore:

$$\begin{aligned}(y^2 + x^2)(y - x) &> 0(y^2 + x^2)\\ y^3 + yx^2 - xy^2 - x^3 &> 0\\ y^3 + yx^2 &> x^3 + xy^2\end{aligned}$$

□

Week 4

Learning Objectives:

- Correctly follow a sequence of justified steps to reach a conclusion statement.
- Prove a conclusion statement by first assuming it is false.
- Describe inductive steps.
- Understand logical arguments and apply basic concepts of formal proof.

Essential Reading

- Rosen, K.H. Discrete mathematics and its applications. (New York: McGraw-Hill, 2012) 7th edition, Chapter 1.7, pp.76–83.

2.301 Proof by induction

Mathematical induction is a useful proof method that has several steps that must be followed. We can consider mathematical induction as a row of standing dominoes and we want to prove that all dominoes fall.

In order to prove that all dominoes fall, we must first and foremost prove that the first domino falls. After that we prove that if one domino falls, the next one must also fall.

Mathematically, if $P(0)$ is true and $\forall k \in \mathbb{N} P(k) \rightarrow P(k+1)$, then we can conclude that $\forall n \in \mathbb{N} P(n)$ is true.

Proof by induction has three important steps:

- The *Base Case* or *Basis*

  Here we prove that $P(0)$ is true. This allows us to prove that the theorem **starts** true.

- The *Inductive Step*

  Here we prove that $P(k) \rightarrow P(k+1)$. Note that we never assume this to be true. We must always carefully prove it. Because we're trying to prove an implication, we assume $P(k)$ to be true and prove $P(k+1)$ to be true when $P(k)$ is true. The reason for this is that if $P(k)$ is false, then the implication is true anyway. The assumption that $P(k)$ is true is called the *Inductive Hypothesis*.

- The *Conclusion by Induction*

  We finish the proof by writing $\therefore \forall n \in \mathbb{N} P(n)$ is true.

It's common practice to end a proof with the $\square$ symbol. Referred to as **QED** (from Latin *quod erat demonstrandum*).

2.303 Example of a correct proof

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Theorem 9. *The sum of the first n powers of 2, is* $2^n - 1$.

Proof. Let $P(n) = 2^0 + 2^1 + \ldots + 2^{n-1} = 2^n - 1$, prove that $P(n)$ is valid for all n.

Basis: Prove that $P(1)$ is true.

$$\begin{aligned} P(1) &= 2^{1-1} \\ &= 2^0 \\ &= 1 \\ &= 2^1 - 1 \end{aligned}$$

Indutive Step: Prove that $P(k) \rightarrow P(k+1)$ is true.

Assuming $P(k) = 2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$ to be true, prove that $P(k+1) = 2^0 + 2^1 + \ldots + 2^{k-1} + 2^k = 2^{k+1} - 1$ is also true.

$$P(k+1) = 2^0 + 2^1 + \ldots + 2^{k-1} + 2^k$$

By Inductive Hypothesis we know that $2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$, therefore:

$$\begin{aligned} P(k+1) &= 2^k - 1 + 2^k \\ &= 2^k + 2^k - 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

Conclusion: $P(k+1)$ is true, $\therefore \forall n \in \mathbb{N} 2^n - 1$ is true. $\square$

$\forall n\, n < 3^n$

Theorem 10. *$n < 3^n$, for all $n \in \mathbb{N}$*

Proof. Let $P(n) = n < 3^n$, prove by induction that $P(n)$ is true for all n.

Basis: Prove $P(0)$ is true.

$$\begin{aligned} P(0) &= 0 < 3^0 \\ &= 0 < 1 \end{aligned}$$

Inductive Step: Prove $P(k) \to P(k+1)$.

Assuming $P(k) = k < 3^k$ to be true, prove that $P(k+1) = (k+1) < 3^{k+1}$ is true.

$$\begin{aligned} k &< 3^k \\ k+1 &< 3^k + 1 \\ k+1 &< 3^k + 1 < 3^k + 3^k + 3^k \\ k+1 &< 3^k + 1 < 3 \cdot 3^k \\ k+1 &< 3^k + 1 < 3^{k+1} \\ k+1 &< 3^{k+1} \end{aligned}$$

Conclusion: $P(k+1)$ is true, $\therefore \forall n \in \mathbb{N} n < 3^n$ is true. □

2.305 Example of an incorrect proof

We're going to see how easy it is to make a mistake in a proof if we don't follow the steps correctly.

$n + 1 < n \forall n \in \mathbb{N}$

Theorem 11. *$n + 1 < n$, for all $n \in \mathbb{N}$.*

Proof. INCORRECT!!!

Let $P(n) = n + 1 < n \forall n \in \mathbb{N}$.

Prove $P(k) \to P(k+1)$. Assuming $P(k)$ is true, so $k + 1 < k$. Show $P(k+1)$ is true. Adding 1 to both sides of the inequality we get $k + 1 + 1 < k + 1$. Let $l = k + 1$ we get $l + 1 < l$, so $P(k+1)$ is also true. Therefore $P(n)$ is true. □

In this proof, we didn't prove the base case, so the proof in invalid.

2.401 Conclusion

We have learned about several powerful proof techniques. We explored Proof by Induction, which exploits the fact that natural numbers are like a chain.

We have also seen how contrapositive proofs are, sometimes, easier then proving the original statement. We have also witnessed how proofs can go wrong if we miss an important step.

Week 5

Learning Objectives

- Explore finite or countable discrete structures in the context of computer science.
- Consider how different rules can be applied to appreciate the number of possible outcomes for an event.
- Explore relationships between sets and elements within or across sets.
- Consider how elements in a set can be counted.

3.01 Introduction

During this topic, we study key principles in counting. We study the Pigeon-hole principle and learn to apply to prove theorems.

3.101 Counting

How many outfits can we pick from a collection of 5 pairs of trousers and 7 shirts? Essentially this translates to:

$$\begin{aligned}\binom{7}{1} \cdot \binom{5}{1} & \\ &= \frac{7!}{1! \cdot (7-1)!} \cdot \frac{5!}{1! \cdot (5-1)!} \\ &= \frac{7 \cdot 6!}{6!} \cdot \frac{5 \cdot 4!}{4!} \\ &= 7 \cdot 5 \\ &= 35\end{aligned}$$

Product Rule

The product rule says that if a job can be split into two separate tasks, if there are n ways of doing task 1 and m ways of doing task 2 then the job can be done in $n \cdot m$ ways.

A generalization of this is to state that if there are k tasks and each task can be achieved in n_i ways, then there are $n_1 \cdot n_2 \cdot \ldots \cdot n_k$ ways of achieving the task.

1. Example 1

   How many strings of length 5 can we make with uppercase English letters?

   We have 26 uppercase letters in the English alphabet and there are no restrictions to repetition. For each letter we have 26 options, therefore we can make as many as $26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 = 26^5 = 11881376$.

2. Example 2

   How many strings of length 5 can we make with 3 uppercase English letters and 2 digits?

   This is going to be $26 \cdot 26 \cdot 26 \cdot 10 \cdot 10 = 26^3 \cdot 10^2 = 1757600$

Sum Rule

The sum rule states that if a job can be done in n ways **or** m ways, then it can be done in $n + m$ ways.

1. Example 1

   A teacher is choosing a student to be her assistent from 5 different classes. The classes contain 28, 21, 24, 25, and 27 students. How many possible ways are there to pick an assistant?

   There are $28 + 21 + 24 + 25 + 27 = 125$ ways of picking an assistant.

3.102 Complex counting

Continuing with counting, looking at more advanced techniques.

Example 1

For most accounts, you need to choose a password. In this example, the password must be five to seven characters long. Each drawn from uppercase letters or digits. The password must contain at least one upper case letter.

Let's split this work by length:

Passwords	Length 5	Length 6	Length 7
All passwords (1)	36^5	36^6	36^7
No Letters (2)	10^5	10^6	10^7
Valid Passwords (1 - 2)	60 366 176	2 175 782 336	78 354 164 096
		Total	80 590 312 608

Subtraction Rule

The subtraction rule applies when lists have items in common. This rule is also known as *Inclusion-Exclusion Principle*.

This rule states that if a choice can be made from two lists containing n and m items, then the number of ways to make a choice from these two lists is $n + m -$ items in common.

1. Example 1

   How many integers less than 100 are divisible by either 2 or 3. In other words, we're talking about the cardinality of the union of the set of numbers divisible by 2 and less than 100 and the set of numbers divisible by 3 and less than 100.

   However, this would be cumbersome to calculate. A simpler way is to first calculate how many numbers between 1 and 99 are divisible by 2 using $\lfloor \frac{99}{2} \rfloor = 49$. Similarly, we can calculate how many numbers between 1 and 99 are divisible by 3 using $\lfloor \frac{99}{3} \rfloor = 33$.

   We must remember to decrement numbers that are divisible by both 2 and 3. Such numbers are divisible by 6, therefore $\lfloor \frac{99}{6} \rfloor = 16$.

   So the answer to our original question is $49 + 33 - 16 = 66$.

3.201 The Pigeonhole Principle

The Pidgeonhole Principle states that if n items are put into m containers, with $n > m$, then at least one container must contain more than one item. The Pidgeonhole Principle is also know as *Dirichlet's drawer principle*.

The generalized pigeonhole principle states that:

Theorem 12. *If there are N objects to be placed in k boxes, then at least one box contains the $\lceil \frac{N}{k} \rceil$ objects.*

Proof. **By contradiction**

Assume **none** of the boxes contains more than $\lceil \frac{N}{k} \rceil - 1$ objects. Since we have k boxes, we can conclude that Number of Objects $\leq k(\lceil \frac{N}{k} \rceil - 1) < k(\frac{N}{k} + 1 - 1) = N$.

From that we conclude that Number of Objects $\leq N$, which contradicts our original statement that there are exactly N objects. □

Example 1

How many cards from a standard deck of 52 cards must be selected to guarantee that 5 cards are from the same suit?

There are 4 suits in a standard deck of cards. If we pick 16 cards and spread them evenly among the suits, we will end up with 4 cards for each suit. At the moment we pick the 17^{th} card, it must go to one of the 4 suits, therefore giving 5 cards from the same suit.

We can verify this with $\lceil \frac{17}{4} \rceil = 5$.

3.202 The Pigeonhole Principle: examples

Example 1

In a group of 4 integers, show that there are at least two with the same remainder when divided by 3.

There are exactly three possible remainders when dividing numbers by three. Either the number is divisible by 3, giving a remainder of zero, or it is 1 above a multiple of three, or 2 above a multiple of three.

In any group of 4 integers, we will have at least two numbers with the same remainder when divided by three. This means that we have three boxes (the three possible remainders) and 4 objects (our 4 randomly selected integers). By the pidgeonhole principle, we know that at least one box will contain more than one object.

Example 2

A bag contains 7 blue balls and 4 red balls, how many must be selected to guarantee that three balls are of the same color.

In this case, there are 2 boxes (the colors) and 11 objects. In the worst case, we pick colors evenly. Assuming we have picked 4 balls (2 blue and 2 red), when we pick the 5^{th} ball, it must be either blue or red, therefore giving us our 3 balls of the same color.

In other words, we want to find the number which satisfies $\lceil \frac{x}{2} \rceil = 3$.

Example 3

Select 5 integers from the set $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$; show that at least two integers add up to 9.

The pairs making up 9 are $(1, 8)$, $(2, 7)$, $(3, 6)$, $(4, 5)$. If we label those pairs A, B, C, D, we can see that all numbers in the set belong to one of 4 boxes.

There are 4 boxes and 5 objects, therefore at least one box will have more than 1 object.

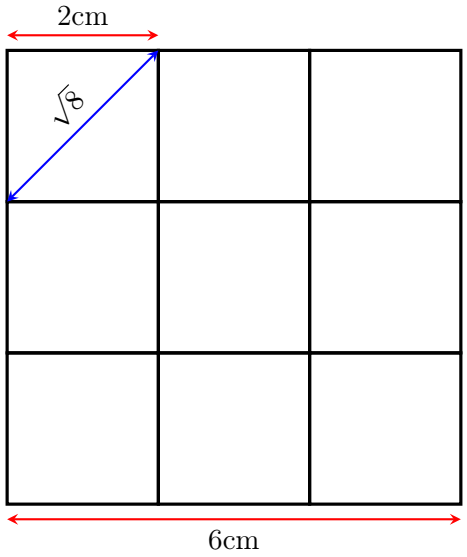
Example 4

There are n people in the room; every pair is either friends or not friends. Show that there are at least two people with the same number of friends.

We know that there are n in the room and the amount of friends people can have are limited to $1, 2, 3, \ldots, n-1$ or $0, 1, 2, \ldots, n-2$. In both of these cases we will have $n-1$ boxes and n people. Consequently, at least one box will have more than one person.

Example 5

Show that if there are 10 dots on a square of $6cm \times 6cm$, there are at least two dots within $\sqrt{8}$ cm.



We can see that a $6cm \times 6cm$ square divided into 9 equal smaller squares of $2cm \times 2cm$. The smaller squares have a hypotenuse of $hyp = \sqrt{2^2 + 2^2} = \sqrt{8}$ (indicated by the blue line).

We have 9 boxes that are $\sqrt{8}$ apart, but have 10 objects. Therefore at least two will be within $\sqrt{8}$ distance from each other.

Week 6

Learning Objectives

- Explore finite or countable discrete structures in the context of computer science.
- Consider how different rules can be applied to appreciate the number of possible outcomes for an event.
- Explore relationships between sets and elements within or across sets.
- Consider how elements in a set can be counted.

Essential Reading

- Sipser, M. Introduction to the theory of computation. (Boston: Cengage Learning, 2014) 3rd edition, Chapters 1.1 and 1.2, pp.31–63
  - PDF

3.301 Permutations

Permutation relates to the arrangement of objects where order **does** matter. For example, how many ways are there for 5 people to form a queue? For the first position in the queue, we can have any of the five people, for second position we can have one out of four people; for the third position, we can have one out of three people and so on. The answer here is that there are $5! = 120$ ways for 5 people to form a queue.

Definition 1 (Permutation)**.** *A permutation of a set of distinct objects is an ordered arrangement of these objects.*

There are $n!$ permutations of n objects. This is a simplification of the definition of Permutation of n objects taken r at a time. If we set $k = r$ we get:

$$\begin{aligned} {}^{n}P_{n} &= \frac{n!}{(n-n)!} \\ &= \frac{n!}{0!} \\ &= n! \end{aligned}$$

The r-permutation of a set of n elements is denoted by ${}^{n}P_{r} = P(n, r)$.

Theorem 13. *For two integers $n, r, 0 \leq r \leq n$. There are:*

$$^nP_r = P(n,r) = n(n-1) + \ldots + (n-r+1) = \frac{n!}{(n-r)!}$$

3.304 Combinations

Combination relates to the arrangement of objects where **does not** matter. For example, if we have 4 animals (mouse, cat, dog, rabbit) and we want to take a side-by-side photo of two animals, how many ways can we choose two animals? Note that photos with the same animals in different order count as equivalent ($(cat, dog) \equiv (dog, cat)$).

Definition 2 (Combination)**.** *A combination of a set of distinct objects is an unordered arrangement of these objects.*

There is only **one** combination of n elements. This is because upon shuffling, only the order of elements change, however with combinations the **order does not matter**.

An r-combination of elements of a set is an **unordered** selection of r elements from this set and is denoted by $^nC_r = C(n,r)$.

Theorem 14. *For two integers $n, r, 0 \leq r \leq n$. There are:*

$$^nC_r = C(n,r) = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Example 1

How many hands of 7 cards can be dealt from a standard deck of 52 cards? First we need to figure out if the order matters. In this case it doesn't because being dealt (5, 7) is equivalent to being dealt (7, 5).

$$\begin{aligned}
^nC_r &= \binom{n}{r} \\
&= \binom{52}{7} \\
&= \frac{52!}{7!(52-7)!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{7!45!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} \\
&= 133\,784\,560
\end{aligned}$$

What if we're dealing 45 card hands, instead?

$$\begin{aligned}
{}^{n}C_r &= \binom{n}{r} \\
&= \binom{52}{45} \\
&= \frac{52!}{45!(52-45)!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{45!7!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} \\
&= 133\,784\,560
\end{aligned}$$

This shows that $C(n, r) = C(n, n - r)$.

Example 2

How many ways are there to choose 11 players from a group of 16 in order to form a team? First we need to decide if the order matters. In case, it doesn't. Therefore it's a combination.

$$\begin{aligned}
{}^{n}C_r &= \binom{n}{r} \\
&= \binom{16}{11} \\
&= \frac{16!}{11!(16-11)!} \\
&= \frac{16 \cdot 15 \cdot 14 \cdot 13 \cdot 13 \cdot 12 \cdot 11!}{11! \cdot 5!} \\
&= \frac{16 \cdot 15 \cdot 14 \cdot 13 \cdot 13 \cdot 12}{5!} \\
&= 4368
\end{aligned}$$

Example 3

How many binary words of length 8 contain equal number of zeroes and ones?

Does the order matter? No. Therefore it's a combination.

$$
\begin{aligned}
{}^{n}C_r &= \binom{n}{r} \\
&= \binom{8}{4} \\
&= \frac{8!}{4!(8-4)!} \\
&= \frac{8!}{4!4!} \\
&= \frac{8 \cdot 7 \cdot 6 \cdot 5 \cdot 4!}{4!4!} \\
&= \frac{8 \cdot 7 \cdot 6 \cdot 5}{4!} \\
&= 70
\end{aligned}
$$

Example 4

How many binary words of length 8 contain at most 3 ones?

At most means 0, 1, 2, or 3. This can be solved as a sum of combinations:

$$
\begin{aligned}
&= \binom{8}{0} + \binom{8}{1} + \binom{8}{2} + \binom{8}{3} \\
&= \frac{8!}{0!8!} + \frac{8!}{1!7!} + \frac{8!}{2!6!} + \frac{8!}{3!5!} \\
&= 1 + 8 + 28 + 56 \\
&= 93
\end{aligned}
$$

Note that $0! = 1$.

Example 5

How many binary words of length 8 contain at least 5 ones?

At least means 5, 6, 7 or 8. This can be solved as a sum of combinations:

$$
\begin{aligned}
&= \binom{8}{5} + \binom{8}{6} + \binom{8}{7} + \binom{8}{8} \\
&= \frac{8!}{5!3!} + \frac{8!}{6!2!} + \frac{8!}{7!1!} + \frac{8!}{8!0!} \\
&= 56 + 28 + 8 + 1 \\
&= 93
\end{aligned}
$$

Week 7

Learning Objectives

- Understand the basic terminologies of automata theory.
- Describe finite automata and what it can represent.
- Build deterministic and nondeterministic finite automata.
- Understand and apply various concepts in automata theory, such as deterministic automata, regular languages and context-free grammar.

4.01 Introduction

We will study simple mathematical machines knows as *Automata*. The next few topics will introduce the prerequisites for understanding automata and later we will learn about how these machines work and process input data.

Riddle:

> There is a farmer who has a mouse, a cat, and a loaf of bread. He has to cross the river from the north to the south by a small boat. He can take up to one of his possessions with him on the boat. The boat cannot operate without the farmer. The cat and the mouse cannot be left alone, as the cast will eat the mouse, and the mouse cannot be left alone with the loaf of bread as it will eat it. How can the farmer cross deliver without losing one of his possessions?

4.101 Basic definitions, letters, strings

An alphabet is denoted by the capital greek letter sigma Σ. If $\Sigma = \{0, 1\}$, we have a **binary alphabet**. If $\Sigma = \{a, b, \ldots, z\}$, then we have a collection of lower case letters.

A *string* or *word* is a finite sequence of letters drawn from an alphabet Σ. Empty strings denoted by ε are strings with zero occurrences of letters from any alphabet.

The length of string x is denoted by $|x|$. For example if $x = 01110101$ then $|x| = 8$. Similarly $|\text{Life is good}| = 12$

Given an alphabet Σ there are a few different things we can do with it:

- The set of **all strings** composed of letters in Σ is denoted by Σ^*

  Note that this is an infinite sequence of possible strings. For example if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon\ 0, 1, 00, 01, 10, 11, \ldots\}$

- The set of **all non-empty strings** composed of letters in Σ is denoted by Σ^+

  Note that this is also an infinite sequence of possible strings. For example if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{0, 1, 00, 01, 10, 11, \ldots\}$

- The set of **all strings of length k** composed of letters in Σ is denoted by Σ^k

  Note that this is a **finite** sequence of possible strings. For example if $\Sigma = \{0, 1\}$, then $\Sigma^2 = \{00, 01, 10, 11\}$. For any alphabet Σ, $|\Sigma^k| = |\Sigma|^{k1}$

A language is a collection of strings over an alphabet. For example the language of palindromes over the binary alphabet $\Sigma = \{0, 1\}$ is $\{\varepsilon, 0, 1, 00, 11, 000, 010, 101, 111, \ldots\}$.

Examples

1. If $\Sigma = \{a, b, c\}$ then what is Σ^2?

   $\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

2. If $\Sigma = \{a, b, c\}$ then what is Σ^1?

   $\Sigma^1 = \{a, b, c\}$

   Note that $\Sigma^1 \neq \Sigma$. Elements in Σ are called **symbols** while elements in Σ^1 are called **strings**, the strings just happen to have length 1.

4.103 What is an automaton?

A *Finite Automaton* is a simple mathematical machine. It's a mathematical model of how computations are performed within **limited memory** space. This machine contains input and output (Reject or Accept).

Each circle in this image is a **state** in the automaton. State A is what we call the **initial state**. An initial state will always have an arrow coming from nowhere. Each of the arrows denote state **transitions** and their labels come from the alphabet dictating what to do next.

For example, if we are at state A and we read a 1, then we go to state B. Similarly, if we are at state A and read a 0, then we go to state C.

[1]The length of the set of all strings of length k from alphabet Σ is equal to the length of the alphabet Σ to the power k.

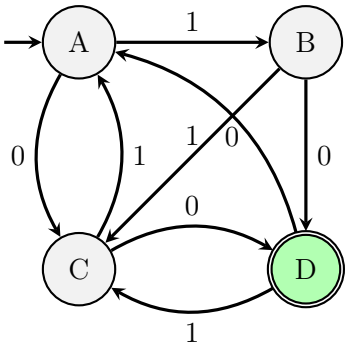


Figure 1: Finite Automaton Example

In this diagram, D is referred to as an **accepting state**. accept states will always be draw as a double circle. What they mean is that if the computation ends at an accept state, then the machine outputs *ACCEPT*.

Formal definition of Finite Automaton

An automaton M is a 5-tuple[2] $(Q, \Sigma, \delta, q_0, F)$ where:

- 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 states**

Using our previous example in figure 1, let's formalise its definition:

- $Q = \{A, B, C, D\}$
- $\Sigma = \{0, 1\}$
- $q_0 = \{A\}$
- $F = \{D\}$
- δ can be represented by the following state transition table

δ	0	1
A	C	B
B	D	C
C	D	A
*D	A	C

[2] A sequence of 5 elements.

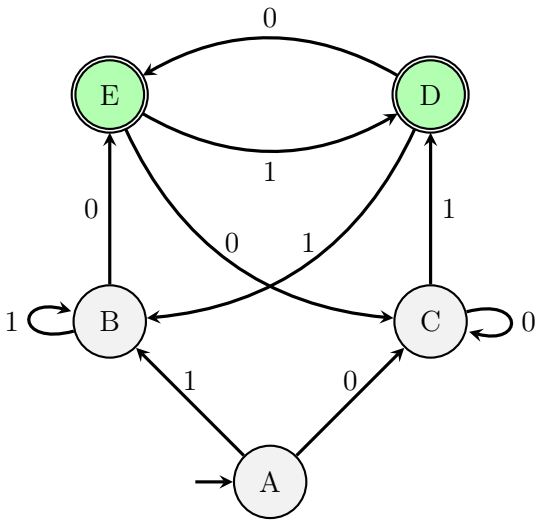


Figure 2: Automaton with 2 inputs

4.201 Finite automata: example

We look at an example of an automaton with 5 states and 2 inputs. We will see if a given input is accepted of rejected by the automaton. Figure 2 has a diagram of this automaton.

- $Q = \{A, B, C, D, E\}$
- $\Sigma = \{0, 1\}$
- $q_0 = A$
- $F = \{D, E\}$
- δ is represented by the table below

	0	1
A	C	B
B	E	B
C	C	D
D	E	B
E	C	D

We give the input 10011 to this automaton and start computation from state A. This can be seen in figure 3.

After processing input 1, we reach B, as can be seen in figure 4. Right after that we process the next input 0 and switch to state E as shows in figure 5. The third bit in the string is another 0, which causes us to switch to state C as in figure 6.

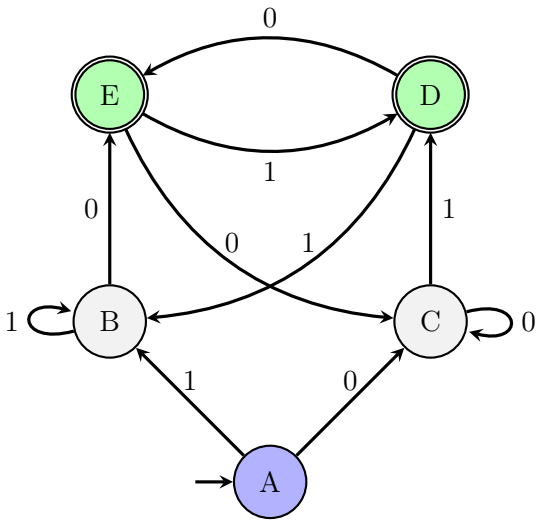


Figure 3: Automaton with 2 inputs: 10011

What follows is input 1 which makes the automaton switch to state D as in figure 7. Finally, we process another 1 which causes the automaton to switch to state B as in figure 8, which is **not** and accept state and, therefore, the output of the computation is *REJECT*.

The next input is 0, which causes to transition from B to E.

One important detail to keep in mind that we don't *ACCEPT* when passing through an accept state, only when the input **ends** at an accept state.

4.202 Language of the automata

Using the same example automaton as before, we can see that that to fall on accept state E, the input must end with 0 (see figure 9). E has two entry points which are B and D.

By looking at the penultimate states, we can see that inputs terminating at the accept state E **must** end with 10.

The only remaining accept state D is symmetrical to C and, therefore, inputs terminating at the accept state D **must** end with 01.

This means that **any** string ending with 01 or 10 will be accepted by our sample automaton.

The set of **all** strings accepted by an automaton is called the **Language** of an automaton. If M is an automaton on alphabet Σ, then $\mathcal{L}(M)$ is the language of M:

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

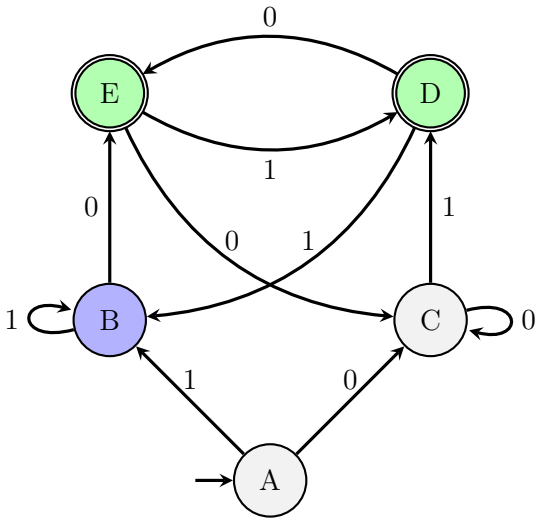


Figure 4: Automaton with 2 inputs: **1**0011

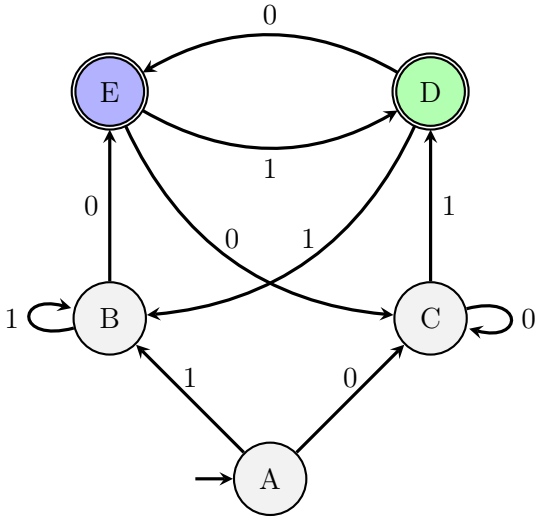


Figure 5: Automaton with 2 inputs: 1**0**011

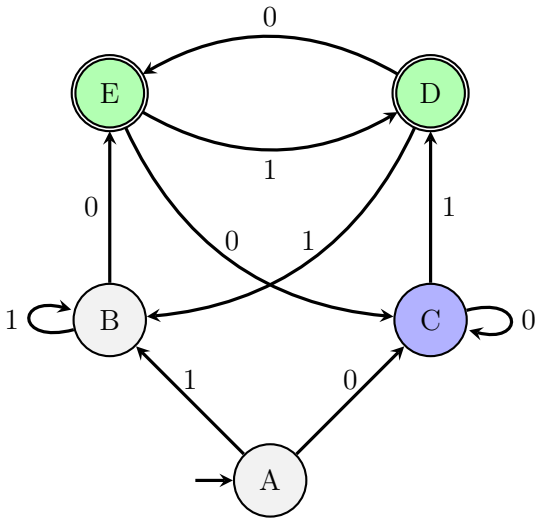


Figure 6: Automaton with 2 inputs: 10**0**11

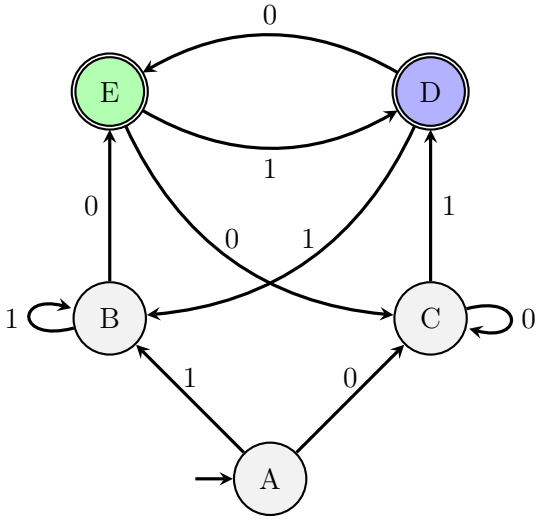


Figure 7: Automaton with 2 inputs: 100**1**1

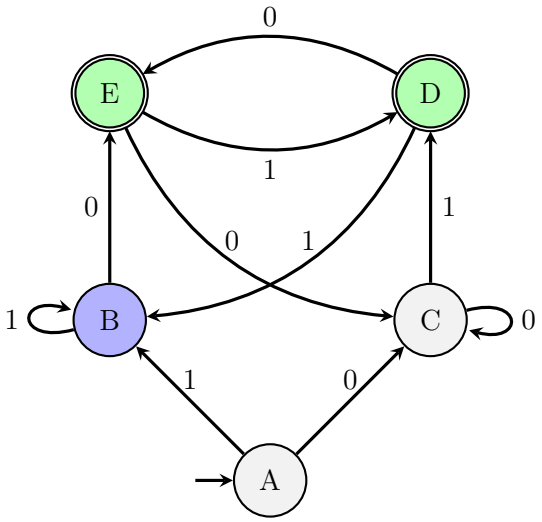


Figure 8: Automaton with 2 inputs: 1001**1**

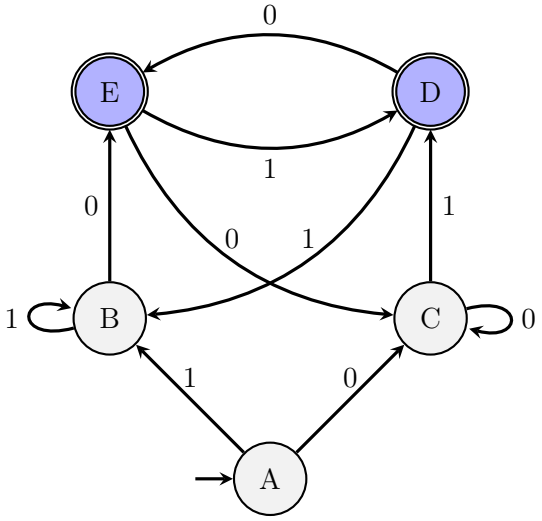


Figure 9: Automaton with 2 inputs: 1001**1**

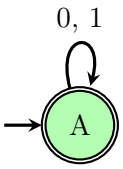


Figure 10: Accepting all binary strings

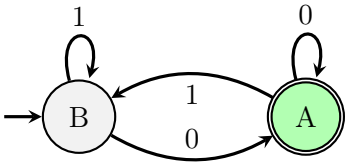


Figure 11: Accepting all binary strings ending with 0

4.204 Recognise a language

Given a set of inputs, can we build an automaton that represents the set of inputs?

Accepting all binary strings

The automaton depicted in figure 10 accepts all binary strings regardless of their length.

Accepting all binary strings ending with 0

The automaton depicted in figure 11 accepts all binary strings ending with 0.

Accepting strings ending 01

The automaton depicted in figure 12 accepts all binary strings containing with 01.

Automaton accepting strings with 00 or 11

This was left as an exercise. My solution is depicted in figure 13.

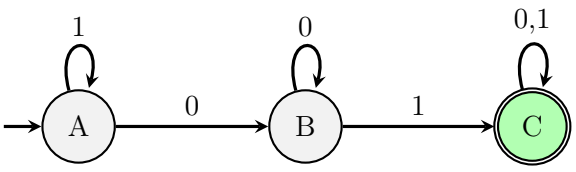


Figure 12: Accepting strings containing with 01

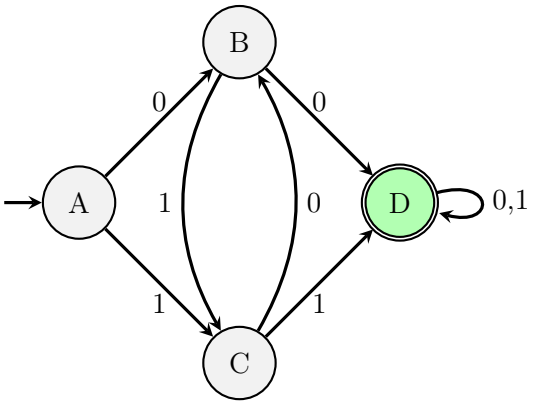


Figure 13: Accepting strings containing 00 or 11

Week 8

Learning Objectives

- Understand the basic terminologies of automata theory.
- Describe finite automata and what it can represent.
- Build deterministic and nondeterministic finite automata.
- Understand and apply various concepts in automata theory, such as deterministic automata, regular languages and context-free grammar.

Essential Reading

- Sipser, M. Introduction to the theory of computation. (Boston: Cengage Learning, 2014) 3rd edition, Chapters 1.1 and 1.2, pp.31–63
  - PDF

4.301 Deterministic finite automata (DFA) vs nondeterministic finite automata (NFA)

Sometimes we can get stuck during computation. Figure 14 shows one such example where we can get stuck depending on the input.

For example, if input is *1100*, we can see that we will start initial state A and loop from A to A after the first input *1*. Then the next *1* will cause us to loop again and remain at state A.

Then we take a *0* which causes us to move to state B. Now, when we take the next *0*, there are no outgoing transitions from B to anywhere else with label *0*, so we get stuck. In the case of figure 14 we get stuck due to a lack of transitions.

Figure 15 has a slightly different problem. Let's work through the states with input *11001*. Again we start at A and take input *1* which causes us to loop back to A. Then read another *1* which, again, loops us back to A. The next input is *0*, which makes us transition to state B. With the next *0* input we have a problem: there are **two** transitions we can take, so which one do we take? We can't make a decision, so we get stuck again. In this case, we get stuck because there are **too many** transitions.

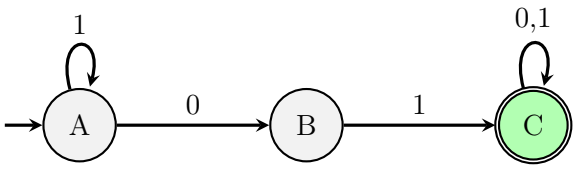


Figure 14: Getting stuck: Not enough Transitions

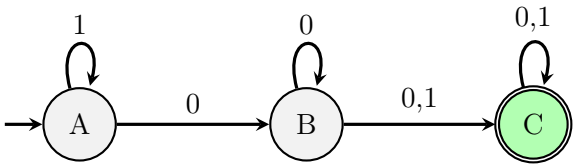


Figure 15: Getting stuck: Too Many Transitions

Deterministic Finite Automata (DFA)

A DFA is the simplest form of Automata. They are *very well behaved* meaning that we jump from state to state deterministically. What this means is that each state has **exactly** one transition for each character of the alphabet and there is a unique start state.

If any of those two requirements are **not** met, then we say the automaton is **non-deterministic**.

Non-deterministic Finite Automata (NFA)

A NFA is just a DFA that breaks at least one of the two DFA requirements. For example, we may encounter a state where, for a given input symbol, we can take one of many alternative paths, or no paths at all.

In the context of NFAs, an input is accepted if there is **at least** one sequence of choices that would lead to an accept state.

Because of all these details, the behavior of an NFA can be more complex than DFA.

NFAs can be used in the implementatio of Regular Expressions.

Figure 16 shows an example NFA. Assuming we have the input *1101*, let's work through the computation. When we read the first symbol *1* we already have two choices: loop back A or transition to B. Assume we take the loop back to A. Another *1* and assume we take the loop again, so we're back at A. The following symbol is *0*, so we must transition to B. Next, we get a symbol *1* and we loop back to B. Because B is not an accept state, the input is rejected.

However, let's try to take another path. With the first symbol *1* we take the transition

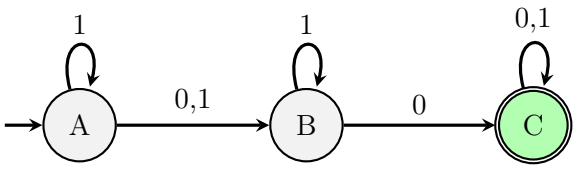


Figure 16: NFA Example

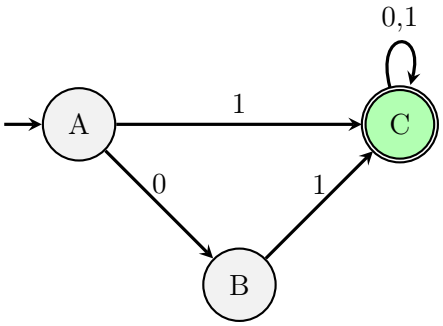


Figure 17: NFA with too few transitions

to B. With the next *1*, we loop back to B. With the following *0* we transition to C and with the final *1* we loop back to C which is an accept state.

4.303 Computation by NFA

What happens with NFAs that have too few transitions? Using figure 17 as an example, let's work the computation of input *001101*.

1. Start at A
2. Read input *0*, transition to B
3. Read input *0*, no transition left. We reject input.

Language of NFA

Again, we're going to use 17 as our example. We would like to study different inputs.

If input starts with *1*, then we transition to accept state C and anything that follows keeps us in state C by looping back.

If the input starts with *01*, then we, again, transtion to accept state C (through B) and anything that follows keeps us in state C by looping back.

Any other strings will cause us to get stuck.

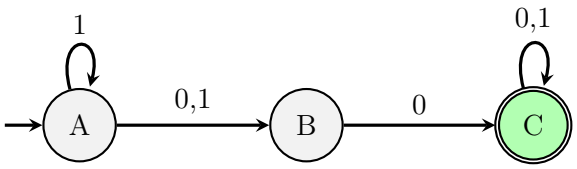


Figure 18: NFA Complex Example

The language of this NFA is, therefore, all binary strings starting with *1* or *01*.

Language of NFA - a complex example

Using figure 18, let's study its language.

If input starts with *0* we transition to B. If another *0* follows we transition to accept state C. Anything that follows is accepted.

If input starts with *1*, we can either transition B or loop back to A. Assuming we loop back to A, any following symbol causes us to transition to B which must read a *0* input to transition to accept state C. Anything that follows is accepted.

The language of this NFA is, therefore, all binary strings starting with *00* or at least one *1* followed by *0*.

Week 9

Learning Objectives

- Describe formal languages in the context of regular expressions.
- Identify examples of regular expressions and finite automaton.
- Write regular expressions with and without the use of finite automaton.

Essential reading

- Sipser, M. Introduction to the theory of computation. (Boston: Cengage Learning, 2014) 3rd edition, Chapter 1.3, pp.63–76
  - PDF

5.101 Regular expressions

Before moving on, we review some basic notions and definitions that we will need:

- an **alphabet** Σ is a non-empty set of *symbols*
- a **string** or word is a finite sequence of symbols drawn from an alphabet
- empty strings are denoted by ε
- the set of all strings composed from symbols in Σ is denoted by Σ^*
- a language is a collection of strings over an alphabet

Regular expressions are designed around *Regular Operations*. There are three operations at our disposal:

- **Union** $\cup$

  $L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$
- **Concatenation** $\circ$

  $L_1 \circ L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$
- **Star** $*$

  $L_1^* = \{x_1 x_2 \dots x_m \mid m \geq 0, x_i \in L_1\}$

Example

$$
\begin{aligned}
A &= \{red, green, pink\} \\
B &= \{apple, banana, kiwi\} \\
A \cup B &= \{red, green, pink, apple, banana, kiwi\} \\
A \circ B &= \{redapple, redbanana, redkiwi, greenapple, greenbanana, \\
&\quad greenkiwi, pinkapple, pinkbanana, pinkkiwi\} \\
A^* &= \{\varepsilon, red, green, pink, redred, redgreen, \\
&\quad redpink, greenred, greengreen, greenpink, \\
&\quad pinkred, pinkgreen, pinkpink, \ldots\}
\end{aligned}
$$

Properties of regular operations

Operation	Property	Example
Union	Commutative	$A \cup B = B \cup A$
	Associative	$(A \cup B) \cup C = A \cup (B \cup C)$
	Identity	$A \cup \emptyset = A$
	Idempotence	$A \cup A = A$
	Distributive	$(A \cup B) \circ C = (A \circ C) \cup (B \circ C)$
Concatenation	Associative	$(A \circ B) \circ C = A \circ (B \circ C)$
	Identity	$A \circ \varepsilon = A$
	Domination	$A \circ \emptyset = \emptyset$
	Distributive	$A \circ (B \cup C) = (A \circ B) \cup (A \circ C)$
Kleene Star		$\emptyset^* = \{\varepsilon\}$
		$\varepsilon^* = \varepsilon$
		$(A^*)^* = A^*$
		$A^*A^* = A^*$
		$(A \cup B)^* = (A^*B^*)^*$

Atomic regular expressions

The emptylanguage $\emptyset$ is a regular expression, which is the empty regular language.

Any symbol a from Σ is a regular expression and its regular language is $\{a\}$.

The empty string ε is a regular expression and its regular language is $\{\varepsilon\}$.

Compound regular expressions

By using the regular operations combined with the atomic regular expressions, we can build up compound regular expressions. The operations preserve the regularity, which means that:

- **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 the regular expression

- What is the language of ab^*?
  - $\{a, ab, abb, abbb, \ldots\}$
- What is the language of $ab^* \cup b^*$?
  - $\{a, ab, abb, abbb, \ldots\} \cup \{\varepsilon, b, bb, bbb, \ldots\} = \{\varepsilon, a, b, ab, bb, abb, bbb, \ldots\}$
- What is the language of $ab^+ \cup b^+b$?
  - $\{ab, abb, abbb, \ldots\} \cup \{bb, bbb, bbbb, \ldots\} = ab^* \cup b^*/\{a, \varepsilon, b\}$

Examples on binary alphabet $\Sigma = \{a, b\}$

- What is the language of Σ^*a?
  - $\{a, aa, ba, aaa, aba, \ldots\}$
  - All strings ending with a
- What is the language of $\Sigma^*a\Sigma^*$?
  - $\{a, aa, ab, ba, aaa, aab, aba, abb, baa, bab, bba, \ldots\}$
  - All strings containing at least one a.

Read regular expressions

- Order of Precedence: $*, \circ, \cup$
- Example: Which is the language of $a \cup bc^*$?
  1. $bcbc$
  2. $accc$
  3. aaa
  4. $bccc$

The answer is **4**: $a \cup bc^* = a \cup b(c^*) = a \cup (b(c^*))$. The language is $\{a, b, bc, bcc, bccc, \ldots\}$.

5.103 Design regular expressions

Now we know all the necessary tools to start designing regular expressions for a given language.

All binary words containing bb

The language of all words containing bb is $\{bb, abb, bba, bbb, aabb, abba, abbb, \ldots\}$. We **know** the word can contain anything before and after bb, including the empty string ε. Therefore the regular expression is given by $(a \cup b)^* bb (a \cup b)^*$, which is equivalent to $\Sigma^* bb \Sigma^*$.

All binary words ending with ab or ba

The language is given by $\{ab, ba, aab, aba, bab, bba, aaab, aaba, \ldots\}$. We can start the word with anything, including ε as long as it ends with either ab or ba.

Therefore, the regular expression is given by $((a \cup b)^* ab) \cup (a \cup b)^* ba)$ which is equivalent to $(\Sigma^* ab) \cup (\Sigma^* ba)$.

All binary words with at most one a

The language is given by $\{\varepsilon a, b, ab, ba, bb, abb, bba, bab, \ldots\}$.

The language can contain any number of b before and after one or zero a.

Therefore, the regular expression is given by $(b^* ab^*) \cup b^*$.

All binary strings of length 3

The language is given by $\{aaa, aab, aba, abb, baa, bab, bba, bbb\}$. This can be expressed by the regular expression $\Sigma\Sigma\Sigma$.

All binary strings of length at least 3

This is the same as previous example followed by empty string or any string.

Therefore, $\Sigma\Sigma\Sigma\Sigma^* \equiv \Sigma\Sigma\Sigma^+$

All binary strings of length at most 3

This contains:

- All binary strings of length 0: ε
- All binary strings of length 1: Σ
- All binary strings of length 2: $\Sigma\Sigma$
- All binary strings of length 3: $\Sigma\Sigma\Sigma$

Now we take the union of it all:

$\varepsilon \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma$

5.201 Regular expressions and finite automata

Kleene's Theorem states that a language is regular **if and only if** it can be described by a regular expression.

The application of **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, then it can be described by a regular expression.

We know that the language of Finite Automata is regular and for every regular language there is a Finite Automaton admitting this language. We also know, from Kleene's Theorem, that a regular language can be expressed by a regular expression.

We can use this notion to establish a *link* between regular expressions and finite automata. Also, we can modify Kleene's Theorem in the following way:

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

Converting Finite Automaton to Regular Expression

We start with a simple two-state Finite Automaton as in figure 19. The first step is to add a new initial state C which a transition ε from C to A.

Then we create a new final state, D, with a transition ε from B to D. Our old states, A and B, became regular states.

From this point on, we start removing states and transitions by making them more complex. For example, we remove state B by noticing that the path ABA is equivalent to 00^*1 and the path ABD is equivalent to 00^*. Therefore, we can remove B and modify

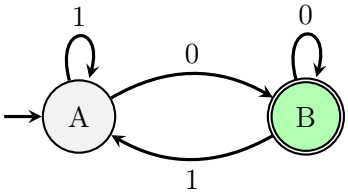


Figure 19: Finite Automaton to Be Converted to Regular Expression

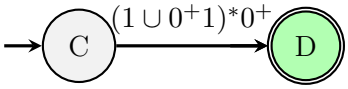


Figure 20: Finite Automaton Converted to Regular Expression

the transitions accordingly. When we have multiple transitions from one state to another, we can take the union of all transitions and collapse into a single transition.

Also note that 00^* is the same as 0^+, so we can can simplify the transition from A to D. Finally, we can remove state A by realising that we have a single path from C to D equivalent to $(1 \cup 0^+1)^*0^+$. The result is shown in figure 20.

5.203 Regular expressions and finite automata: examples

The following figures 21, 22, 23, 24, 25, and 26 show the entire process of converting a finite automaton to regular expression.

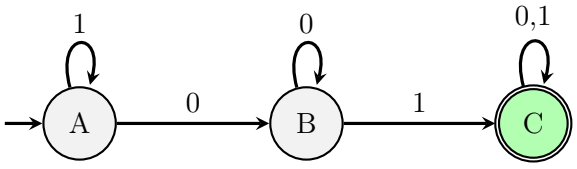


Figure 21: Step 0: Initial Finite Automaton

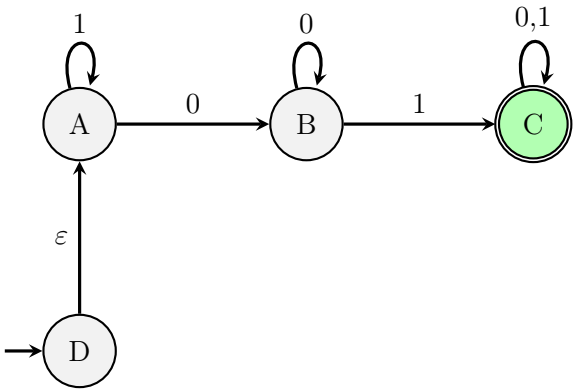


Figure 22: Step 1: Add new initial state

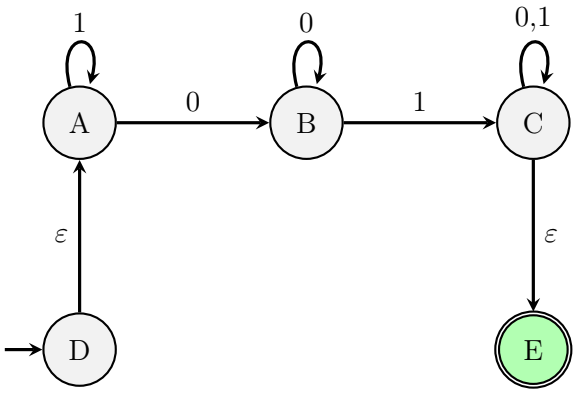


Figure 23: Step 2: Add new final state

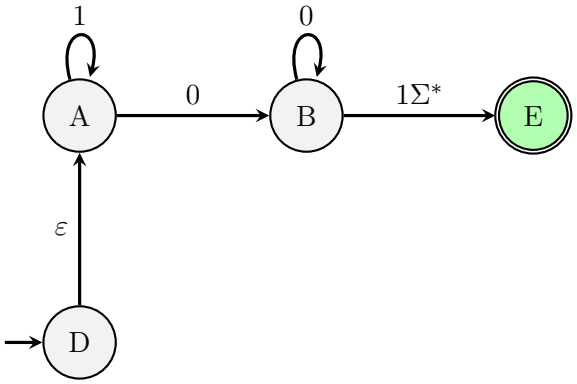


Figure 24: Step 3: Remove state C

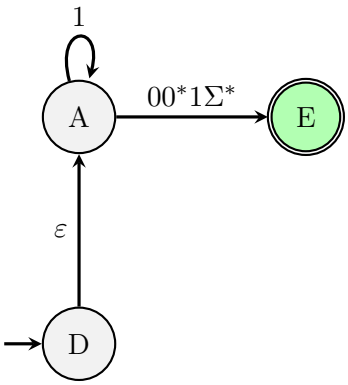


Figure 25: Step 4: Remove state B

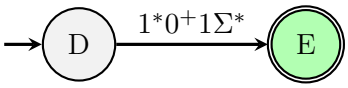


Figure 26: Step 5: Remove state A

Week 10

Learning Objectives

- Describe formal languages in the context of regular expressions
- Identify examples of regular expressions and finite automaton.
- Write regular expressions with and without the use of finite automaton.

Essential Reading

- Sipser, M. Introduction to the theory of computation. (Boston: Cengage Learning, 2012) 3rd edition, Chapter 1.4, pp.77–82.
  - PDF

5.301 Regular or non-regular?

A language is referred to as *regular* if it can be accepted by a finite automaton. Moreover, regular languages can be accepted by regular expressions. Every finite language is regular.

The problem here is that building finite automatons and regular expressions is a non-obvious task; therefore we must implement other techniques such as breaking a language into smaller regular languages and building the language back up by relying on properties of regular languages.

Closure properties

Theorem 15. *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
- $L_1 \cup L_2$*: The union of* L_1 *and* L_2
- $L_1 \cap L_2$*: The intersection of* L_1 *and* L_2
- L_1L_2*: The product of* L_1 *and* L_2
- $L_1{}^*$*: The Kleene star of* L_1

How can we show a language is non-regular?

A non-regular language **cannot** be accepted by a finite automaton, however we cannot test **all** finite automata. Likewise, we cannot test all regular expressions.

Example of non-regular languages

- $L = \{a^n b^n \mid n \in \mathbb{N}\}$

  Any number of a followed by any number of b

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

  The language of all binary strings concatenated with itself

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

  The language of all binary strings concatenated with its reverse

- $L = \{a^{n!} \mid n \in \mathbb{N}\}$

  The language composed of all strings in the form of n-factorial numbers of a

- $L = \{a^{n^2} \mid n \in \mathbb{N}\}$

  All the strings in the form of perfect square number of a

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

  All the strings in the form of prime number of a

Using closure properties - intersection

One powerful technique to show that a language is non-regular is by employing closure properties.

Prove $L = \{x \in \{a, b\}^* \mid \#\text{a in x} = \#\text{b in x}\}$ is not regular. Let's list a few strings in this language:

$$L = \{ab, aabb, abab, abba, baab, \ldots\}$$

Proof. By contradiction.

Let's assume L to be regular. We know that $L' = \{x \in a^*b^*\}$ is regular. We also know that the intersection of two regular languages is also regular, therefore $L \cap L' = \{a^n b^n \mid n \in \mathbb{N}\}$ must be regular. We know this language to not be regular, therefore L cannot be a regular language. □

Using closure properties - complement

Prove $L = \{a^i b^j \mid i, j \in \mathbb{N}, i \neq j\}$ is not regular.

Looking at a few example strings from this language:

$$L = \{abb, aab, abbb, aaabb, \ldots\}$$

Proof. By contradiction.

Let's assume L to be regular, therefore $\neg L$ must also be regular.

We know $\neg L = \{a^n b^n\} \cup$non-bitonic is regular. We also know $L' = \{x \in a^* b^*\}$ is regular. Moreover, we know that the intersection of two regular languages is also regular, therefore $\neg L \cap L'$ must be regular.

However, $\neg L \cap L' = \{a^n b^n \mid n \in \mathbb{N}\}$ is non-regular. □

5.303 Pumping lemma

The Pumping Lemma explains a key property of regular languages. The essence of the pumping lemma is about finding a loop or repeated substrings.

Week 11

Learning Objectives

- Consider context-free grammar and it utility in computer science.
- Explore the language of grammar and designing a grammar.
- Be able to convert between regular expressions and context-free grammar.
- Understand conversion to normal form.
- Understand Chomsky Normal Form.

6.01 Introduction

We discuss Grammars, which is a set of rules which recursively defines the structure of strings.

6.101 Grammar

As we've seen before, regular languages can be implemented by Finite Automata while irregular languages, cannot.

When we want to generate irregular languages, we must employ a more powerful technique called Context-free Grammar. Simply put, a grammar is a set of rules for connecting strings together. They describe the structure of strings recursively.

A Context-free grammar is a type of grammar where rules are independent of the context in which variables occur.

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

Variables (V) Also referred to as non-terminals. A finite set of symbols usually denoted by capital letters.

Terminals (Σ) A finite set of symbols denoted by capital sigma.

Rules (R) A finite set of mappings from a variable to a string consisting of variables and terminals.

Start Variable (S) A member of V, positioned on the left-hand side of the top rule.

Example

$$S \to bSa$$
$$S \to ba$$

This grammar has two production rules. S is the non-terminal and the start variable. There are two terminals a and b.

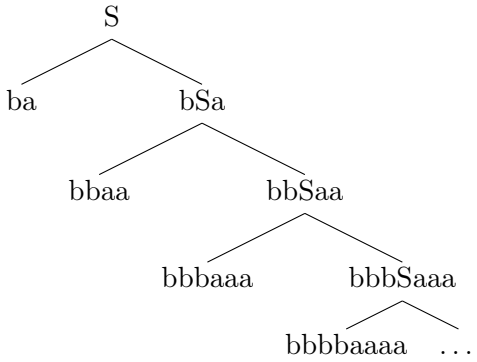
Generating strings

1. Starting from **start symbol**, read its rule
2. Find a **variable** in the rule of the start symbol and replace it with the **rule** for that variable
3. Repeat step 2 until there are no variables left

A few notes:

- A **derivation** is a sequence of substitutions in generating a string
- There may be more than one rule for a variable. We can use the symbol | to indicate *or*: $S \to bSa \mid ba$

With this rule, we have a recursively generated string that looks like so:



We say that **u derives v** or $u \Rightarrow^* v$ if there is a derivation from u to v.

Example

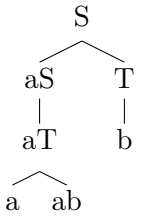
$$S \to aS \mid T$$
$$T \to b \mid \varepsilon$$

Variables S, T

Terminals a, b

Starting variable S

We want to find 3 strings derived from S. We can use the following derivation tree.



Of course, there are many more strings that can be generated from this grammar. For example the S in the first aS, could be replaced with aaS and so on.

Let's find strings which can't be generated from this grammar:

bb b comes from T which comes from S. To get bb we would need to have SS somewhere. But there's no way to produce it.

abb ab comes from aS, which produces aT which produces ab. To produce abb we would have to produce aTT somewhere.

aba Similar case to previous one.

6.201 Language of a grammar

The language of a grammar is the set of all strings that can be generated from the starting symbol using the rules of the grammar.

If $G = (V, \Sigma, R, S)$, then $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

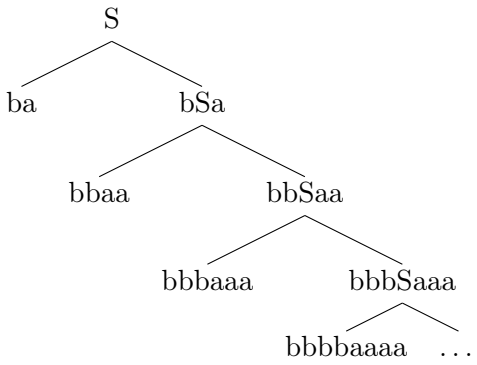
The language of any context-free grammar, is context-free.

Example

What is the language of the following grammar G_1:

$$S \to bSa$$
$$S \to ba$$

If we produce a few strings from this grammar, we can build the following tree:



It becomes clear that all strings will have the same amount of b and a, always starting with b. So the language is $\mathcal{L}(G_1) = \{b^n a^n \mid n \geq 1\}$.

Writing the grammar as a 4-tuple, we have $G_1 = (\{S\}, \{a, b\}, \{S \to bSa, S \to ba\}, S)$.

6.203 Designing a grammar

One technique that can be used when desining a context-free grammar for a context-free language, is to decompose the strings of the language to find a recursive relation in the structure of the language, from this recursive relation we build the grammar.

Decompose and build recursively

The language of palindromes over a binary alphabet is not regular, but it is context-free.

A palindrome is a word that reads the same from either side. This means that the first and last letters are the same, and that the second the second-to-last letters are the same, and so on.

Note that the empty string is a palindrome, so is a single letter by itself, a detail missed during our lectures which is confirmed by Hopcroft figure 5.1.

Assuming the binary alphabet to be $\Sigma = \{a, b\}$, we can say that aa, and bb are palindromes. So are aba, aaa, bab, bbb, and so on.

We can start converting this into a grammar for a palindrome W. If the first letter is a, then we have the rule $S \to aAa$ and if the first letter is b, we have the rule $S \to bAb$. Also, the empty string is a palindrome, which gives us $S \to \varepsilon$.

A is a palindrome, so it belongs to our language, therefore, it could be S. This gives us the rule $A \to S$. We can simplify this grammar into the following:

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

Checklist

After designing a grammar, we can apply this checklist:

Consistency if all strings generated by the grammar fit the description of the language

Completeness all strings in the description can be generated by the grammar

Terminating recursion all recursions used in the grammar terminate

Decompose and build recursively

Build a grammar for the language of all binary strings with even number of zeros.

If the first letter is 1, then whatever follows must have an even number of zeros: $S \to 1A$.

If the first letter is 0, then we must have another 0 after a number of characters, so we have 0A0B, but both A and B must have even number of zeros, meaning they belong to the language and can be decomposed as S: $S \to 0S0S$.

The empty string also has even number of zeros: $S \to \varepsilon$.

Taking the union of these three rules we have:

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

Week 12

Learning Objectives

- Consider context-free grammar and its utility in computer science.
- Explore the language of grammar and designing a grammar.
- Be able to convert between regular expressions and context-free grammar.
- Understand conversion to normal form.
- Understand Chomsky Normal Form.

Essential Reading

- Hopcroft, J., R. Motwani and J.D. Ullman Introduction to automata theory, languages and computation. (Harlow: Pearson Education Ltd, 2013) 3rd edition, Chapter 5, pp.171–224.
  - PDF

6.301 Regular expression to context-free grammar

We have already know that the language of context-free grammars is context-free. We havre also established that regular languages are also context-free. Furthermore, we know that a language is considered regular if and only if it can be expressed as a regular expression. Therefore, we can conclude that **all** regular expressions can be written as context-free grammars.

Note, however, that the reverse of this statement is not always true, since context-free languages are **not** always regular.

Example 1

Let's convert the regular expression ab^* to a Context-free Grammar (CFG).

We can see that the expression b^* can be written as $U \to bU \mid \varepsilon$.

The expression ab^* can be written as $S \to aU$.

The final CFG is:

$$S \to aU$$
$$U \to bU \mid \varepsilon$$

Example 2

Convert the regular expression $ab^* \cup b^*$ to a CFG.

- b^* can be expressed as $U \to bU \mid \varepsilon$
- ab^* can be expressed as $S \to aU$

The final CFG is:

$$S \to aU \mid U$$
$$U \to bU \mid \varepsilon$$

Example 3

Convert the regular expression $ab^+ \cup b^+b$ to a CFG.

- b^+ can be written as $U \to bU \mid b$
- b^+b can be written as $T \to Ub$
- ab^+ can be written as $S \to aU$

The final CFG is:

$$S \to aU \mid T$$
$$T \to Ub$$
$$U \to bU \mid b$$

Which can simplified a little into:

$$S \to aU \mid uB$$
$$U \to bU \mid b$$

Example 4

Convert $\Sigma^* a \Sigma^*$ to a CFG, $\Sigma = \{a, b\}$

- Σ^* can be written as $U \to aU \mid bU \mid \varepsilon$

The final CFG is:

$$S \to UaU$$
$$U \to aU \mid bU \mid \varepsilon$$

Example 5

Binary strings of length at least 3: $\Sigma\Sigma\Sigma^+$

- Σ^+ can be written as $U \to aU \mid bU \mid a \mid b$
- $\Sigma\Sigma^+$ can be written as $V \to aU \mid bU$
- $\Sigma\Sigma\Sigma^+$ can be written as $S \to aV \mid bV$

The final CFG is:

$$S \to aV \mid bV$$
$$V \to aU \mid bU$$
$$U \to aU \mid bU \mid a \mid b$$

6.304 Chomsky Normal Form

A context-free is in Chomsky Normal Form if every rule is of the form:

$$S \to XU$$
$$S \to a$$

- a is a terminal symbol
- X, U are not the start variable
- $S \to \varepsilon$ is permitted if S is the start variable

Conversely, the following grammars are **not** in Chomsky Normal Form:

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

$V \to \varepsilon$ **where** V **is not a start variable** Due to ε rules

$U \to V$ **where** V **is a variable** Due to unit rules

$X \to 1UV$ The length of the rule is > 2. Referred to as **Improper Rule**

We can substitute rules that are not in Chomsky Normal Form with other rules that are in Chomsky Normal Form.

Convert to Chomsky Normal Form

1. Add a new start variable S_0, make a rule $S_0 \to S$
   - ensures that start variable never appears in the right-hand side
2. Eliminate ε rules, $U \to \varepsilon$ where U is not a start variable
   - for each occurence of U on the right-hand side, add a new rule with that particular occurence of U deleted
3. Remove the unit rules $A \to B$
   - Add $A \to x$ for every $B \to x$
4. Convert to proper forms

Example 1

1. Starting CFG

$$
\begin{aligned}
U &\to XUX \mid 0Y \\
X &\to Y \mid U \\
Y &\to 1 \mid \varepsilon
\end{aligned}
$$

2. Add new start variable

$$
\begin{aligned}
S &\to U \\
U &\to XUX \mid 0Y \\
X &\to Y \mid U \\
Y &\to 1 \mid \varepsilon
\end{aligned}
$$

3. Remove ε rule $Y \to \varepsilon$

$$
\begin{aligned}
S &\to U \\
U &\to XUX \mid 0Y \\
X &\to Y \mid U \mid \varepsilon \\
Y &\to 1
\end{aligned}
$$

4. Remove ε rule $X \to \varepsilon$

$$
\begin{aligned}
&S \to U \\
&U \to XUX \mid 0Y \mid UX \mid XU \mid U \\
&X \to Y \mid U \\
&Y \to 1
\end{aligned}
$$

Remove unit rule $X \to Y$

$$
\begin{aligned}
&S \to U \\
&U \to XUX \mid 0Y \mid UX \mid XU \mid U \\
&X \to 1 \mid U \\
&Y \to 1
\end{aligned}
$$

Remove unit rule $U \to U$

$$
\begin{aligned}
&S \to U \\
&U \to XUX \mid 0Y \mid UX \mid XU \\
&X \to 1 \mid U \\
&Y \to 1
\end{aligned}
$$

Remove unit rule $X \to U$

$$
\begin{aligned}
&S \to U \\
&U \to XUX \mid 0Y \mid UX \mid XU \\
&X \to 1 \mid XUX \mid 0Y \mid UX \mid XU \\
&Y \to 1
\end{aligned}
$$

Remove unit rule $S \to U$

$$
\begin{aligned}
&S \to XUX \mid 0Y \mid UX \mid XU \\
&U \to XUX \mid 0Y \mid UX \mid XU \\
&X \to 1 \mid XUX \mid 0Y \mid UX \mid XU \\
&Y \to 1
\end{aligned}
$$

Remove improper rule XUX

$$
\begin{aligned}
S &\to ZX \mid 0Y \mid UX \mid XU \\
U &\to ZX \mid 0Y \mid UX \mid XU \\
X &\to 1 \mid ZX \mid 0Y \mid UX \mid XU \\
Y &\to 1 \\
Z &\to XU
\end{aligned}
$$

Remove improper rule $0Y$

$$
\begin{aligned}
S &\to ZX \mid VY \mid UX \mid XU \\
U &\to ZX \mid VY \mid UX \mid XU \\
X &\to 1 \mid ZX \mid VY \mid UX \mid XU \\
Y &\to 1 \\
Z &\to XU \\
V &\to 0
\end{aligned}
$$

Week 13

Learning Objectives

- Understand the process of computation through Turing machines.
- Consider the design and utility of Turing machines.
- Explore the power and language of Turing machines.
- Discuss non-context-free languages.

Essential Reading

- Forbes, M. A theoretical introduction to Turing Machine. (World Technologies, 2014), Chapter 1, pp.4-21.

7.101 Turing machines

The Turing Machine is a mathematical model of computation proposed by Alan Turing. In its most basic architecture, a Turing Machine is a finite automaton with unbounded Random Access Memory in the form of an infinite tape.

The tape serves as input of instructions and working space for computations. Every cell in the tape contains one character, some cells are empty, those are denoted by a ■. There's a read/write head that moves along the x axis on both directions controlled by the Finite State Automaton. Figure 27 depicts a model of a Turing Machine.

Formaly, a Turing Machine consists of $(Q, \Sigma, \Gamma, \delta, q_1, q_{Acc}, q_{Rej})$, where:

TH A Tape Head

Q A finite set of states

Σ The input alphabet where $\Sigma \subseteq \Gamma$

Γ The tape alphabet, including the blank symbol ■

δ A transition function $\delta : Q \times \Gamma \to (Q \times \Gamma \times \{L, R\})$

q_1 The start state, where $q_1 \in Q$

q_{Acc} Accept state

q_{Rej} Reject state

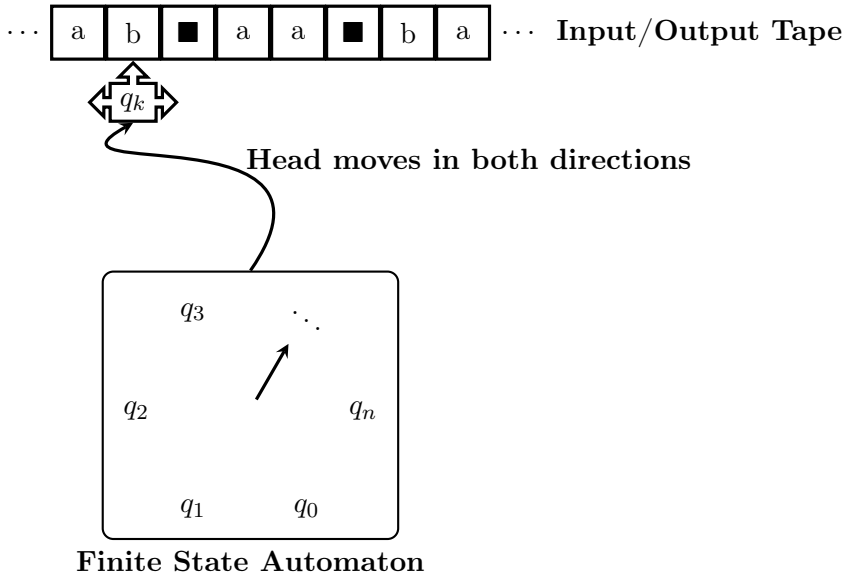


Figure 27: Turing Machine

The transition function δ takes two arguments: a state, and a letter from the alphabet Γ; and returns a state, a letter to be written on the current cell of the tape, the direction to which the tape head should move (L for left, R for right).

Turing machine transitions

Each transition is of the form:

$$letter \rightarrow letter, direction$$

For example: $a \rightarrow b, R$. What this transition means is:

If the letter under the tape head is a, then write b; and move right.

Figure 28 depicts a slightly more complex FSA for a Turing Machine:

We can represent the FSA as a table, shown in table 0.1.

Table 0.1: Turing FSA as a Table

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

DFA vs TM

- TMs may not terminate when the input is completely processed

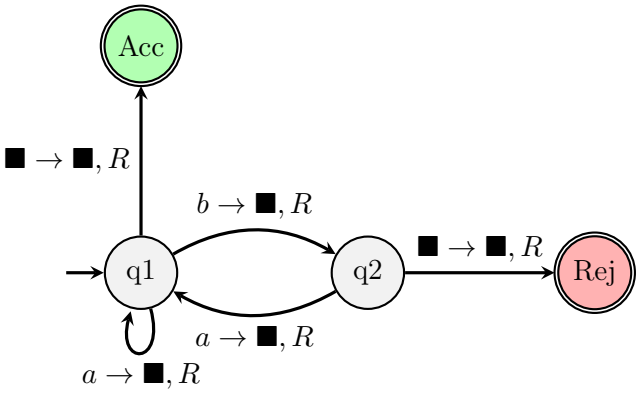


Figure 28: Turing Machine FSA

- TMs may process input multiple times
- TMs terminate at Accept or Reject states
- DFA passing through Accept or Reject does not terminate computation
- TMs may enter infinite loop
- TMs can manipulate the input

Week 14

Learning Objectives

- Understand the process of computation through Turing machines.
- Consider the design and utility of Turing machines.
- Explore the power and language of Turing machines.
- Discuss non-context-free languages.

Essential Reading

- Kozen, D.C. Automata and Computability. (New York: Springer, 2007), Lecture 32, pp.235–238.
  - PDF

7.301 The power of Turing machines

Sorting

Given the input $w \in (1 \cup 0)^*$, sort the input so that $w \in 1^*0^*$.

Note that we're not dealing with accepting or rejecting input, rather we're operating on the input, modifying it.

The procedure for this, goes like so:

1. Parse through the already sorted input of 1*0*
2. Find the first one and flip it to 0
3. Go back to the leftmost 0 and flip it to a 1
4. Goto 2

We can build the following Turing Machine for this procedure:

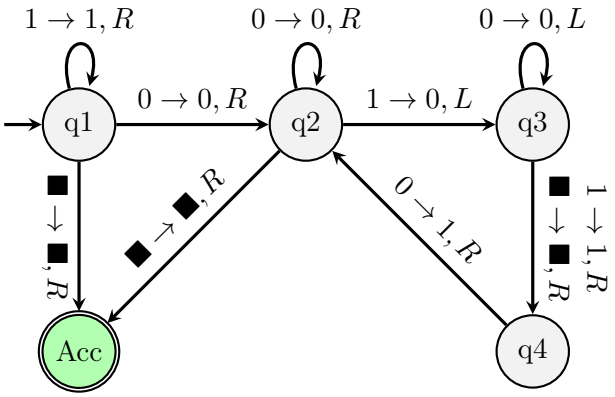


Figure 29: Sorting with Turing Machine

7.303 The language of Turing machines

The Language of TMs

The language of a Turing Machine, M, is composed of all the strings accepted by M. Formally, $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \, accepts \, w\}$.

- If $w \in M$, M reaches *accept* state
- If $w \notin M$, M **does not** reach *accept* state
  - either reaches *reject* state
  - or enters infinite loop
- A language is **recognizable** if it is accepted by a Turing Machine
- A Turing Machine that recognizes $\mathcal{L}(M)$ is called a **Recognizer**
- Recursively Enumerable (RE) is the class of all recognizable languages
- A Turing Machine that never enters an infinite loop is called a **Decider**
- A language accepted by a decider is **Decidable**
- **R** is the class of all decidable languages

Halting problem

- Every decider is a Turing Machine
- $R \subset RE$
- Halting Problem: Given a Turing Machine, does is halt?

- Church-Turing Thesis states that the Halting Problem is an undecidable problem

Hailstone Sequence

```
function hailstone(n)
{
  while (n != 1) {
    if ((n & 1) == 0) {
      n = n / 2
    } else if ((n & 1) == 1) {
      n = 3 * n + 1
    } else {
      return n;
    }
}
```

Nobody knows whether the Hailstone Sequence always terminates. Collatz Conjecture states that it does, but it's yet to be proven.

What we know

- Every regular language is context-free (FSA and RE)
- Every context-free languages is Turing-decidable (CFGs)
- Every decidable language is recognisable (Decide Turing Machine)

Chomsky Hierarchy

Grammar	Languages	Automaton	Example
Type-0	Recursively Enumerable	Turing Machine	
Type-1	Context-sensitive	Turing Machines with bounded tape	$a^n b^n c^n$
Type-2	Context-free	Push-down	$a^n b^n$
Type-3	Regular	Finite State	$a^* b^*$

Week 15

Learning Objectives

- Understand the process of algorithmic thinking and a number of proof techniques, and apply this knowledge to solve a range of computer science problems.
- Explore different techniques for sorting and searching using algorithms.
- Identify the value of using different types of algorithmic techniques in different contexts.

Essential Reading

- Rosen, K.H. Discrete mathematics and its applications. (New York: McGraw-Hill, 2012) 7th edition, Chapter 3.1, pp.191–204

8.101 What is an algorithm ?

An algorithm is a set of steps required to complete a task.

Informal definition

Problem What is the problem we're trying to solve?

Input What information do we have available?

Output What is the output of the algorithm applied to the inputs?

An algorithm is the set of steps required to take the input and achieve the outcome.

Steps from a problem to a working program

- Understanding a **problem**
- **Designing** an algorithm that can solve it
- Checking its **correctness**
- **Analysing** the algorithm (time and space complexity, etc)
- **Implementing** it using various programming languages

- **Testing** the program

A formal definition

An ordered set of unambiguous, executable steps that form a terminating process.

Ordered After each step we know what to do next

Unambiguous Each operation is sufficiently clear

Executable Each operation must be doable (e.g. dividing by 0 is not doable)

Terminating There is a finite number of executions. The process should halt eventually.

8.103 Representing algorithms

Can we agree on one representation?

Algorithms are written by humans. If we didn't agree on a standard way of representing algorithms we would end up with difficulties:

- Unclear language
- Imprecise steps
- Lack of details
- Confusion

To circumvent these, we come up with a standard way of representing algorithms composed of:

Protocols A set of rules for writing algorithms

Primitives A set of well-defined building blocks

Pseudocode

A series of notations that can describe ideas and operations. It's made to be intuitive and informal while also being easily readable by humans.

Algorithms written in pseudocode are not made to processed by machines.

Pseudocode rules are not unique.

Primitives of pseudocode

- Assignment

```
x = 10
sum = a + b
```

- Conditional

```
if (a < b):
    b = b - 2
```

or

```
if (a < b):
    b = b - 2
else:
    b = b + 1
```

- Loops

```
while (raining):
    read a book
```

- Functions

```
def add(a, b):
    a + b
```

Steps for designing an algorithm

1. Understand the problem
2. Draw up a plan to solve it
3. Execute the plan
4. Evaluate the accuracy of the solution

Real techniques

Divide and Conquer Divide problem into smaller problems and solve recursively

Greedy Best solution at each moment without looking at the entire problem

Backtracking Begin with one solution and move to next level. If it's not a good solution, go back to previous step.

8.201 Simple algorithms: insertion sort

```
def insertionSort(array):
    i = 1
    while (i < len(array)):
```

```
        tmp = array[i]
        j = i
        while (array[i] < array[j-1] and (j > 0)):
            j = j - 1

        for k in range(0, (i - j)):
            array[i - k] = array[i - k - 1]
        array[j] = tmp

        i = i + 1
    return array
```

8.203 Simple algorithms: bubble sort

```
def swap(array, i, j):
    x = array[j]
    array[j] = array[i]
    array[i] = x

def bubbleSort(array):
    n = len(array)
    count = 0

    for i in range(0, n - 2):
        for j in range(0, n - 2):
            if (array[j + 1] < array[j]):
                swap(array, j, j + 1)
                count += 1

            if (count == 0):
                break
    return array
```

Week 16

Learning Objectives

- Understand the process of algorithmic thinking and a number of proof techniques, and apply this knowledge to solve a range of computer science problems.
- Explore different techniques for sorting and searching using algorithms.
- Identify the value of using different types of algorithmic techniques in different contexts.

8.301 Binary search

Assuming we have a sorted list, can we do better than looking at every item?

We can employ Binary Search.

What if we use the middle term?

If we have a sorted list and we want to find a particular item, instead of looking at every element, we can look at the midpoint of the list. If that item is less than the one we're looking for, we don't need to look at any other item in the left, thus halving the input size.

Binary search pseudocode

```
def search(List, item):
  If List is Empty:
    Report search has failed
  Else:
    Pivot = middle entry of the list
    If Pivot == Item:
      Report search has succeeded
    Else If Pivot > Item:
      List = Items preceding Pivot
      search(List, item)
    Else:
```

```
12          List = Items following pivot
13          search(List, item)
```

8.304 Heap sort

What is a binary tree

- A rooted tree
- Every vertex has at most 2 children

Heap sort is a sorting algorithm

What is a complete binary tree?

- Every node has exactly two children, except for the last level
- All leaves are placed as far to the left as possible

Depending on the sorting order needed for the array, we can use either Max Heap or Min Heap.

Max Heap

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

Min Heap

- A complete binary tree
- Each internal node has a value **less than or equal to** its children

Heapify

The process of transforming a complete binary tree into a heap tree.

Heap sort

1. Represent list in the form of a complete binary tree

45	21	3	56	10	7	41	9

Figure 30: List to be represented as Binary Tree

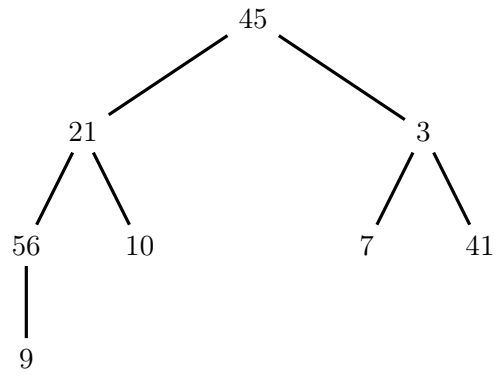


Figure 31: Binary Tree representation of list

2. Heapify the tree

   We traverse the tree from bottom to top. Looking at the two lowest terms, if it's not in Min Heap order, we swap them:

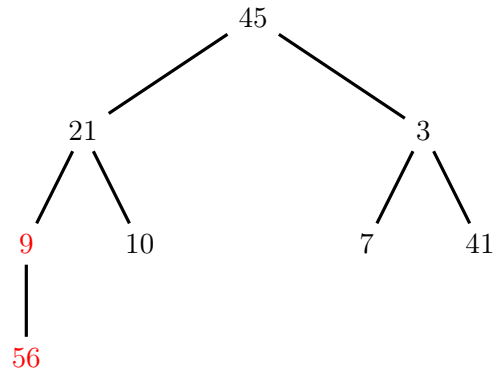


Figure 32: Swap 9 and 56

Once we've done that, we move up the tree. Do 21, 9 and 10 for a Min Heap? No, then we swap:

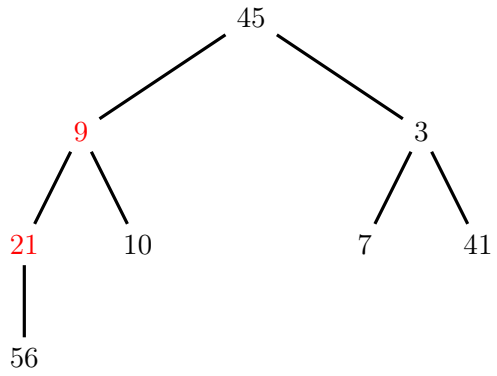


Figure 33: Swap 9 and 21

Moving to the right we look at 3, 7, and 41. Do they form a Min Heap? Yes, we don't need to touch them.

Moving up again we encounter 45, 9, and 3. Do they form a Min Heap? No, then we swap:

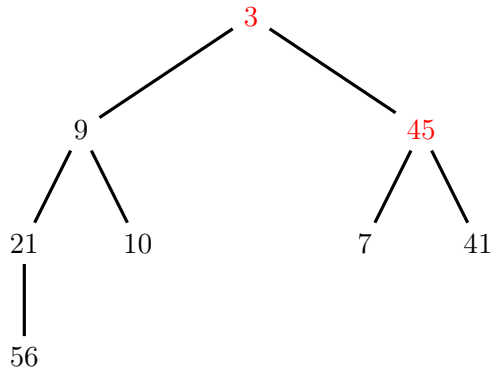


Figure 34: Swap 3 and 45

3. Remove the Node

   Now that the list is heapfied, the root of the tree contains the smallest value, which can be removed and written to a list.

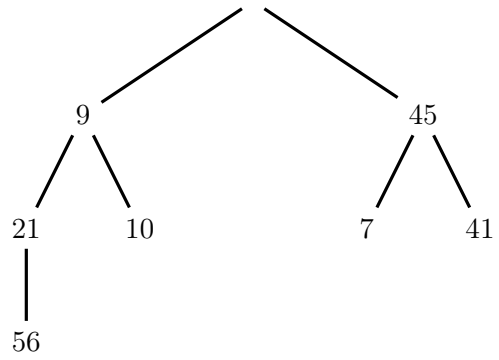


Figure 35: Remove 3

Next we take the last item in the bottom right of the tree and place it in the root:

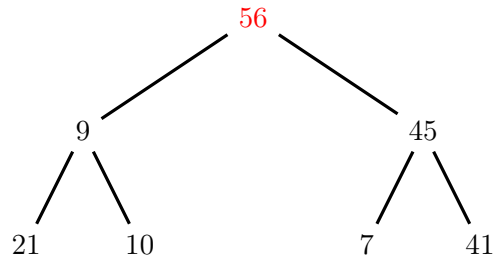


Figure 36: Move 56 to root

The resulting tree is no longer a Min Heap. We must heapify again. The figures that follow, show the rest of the process.

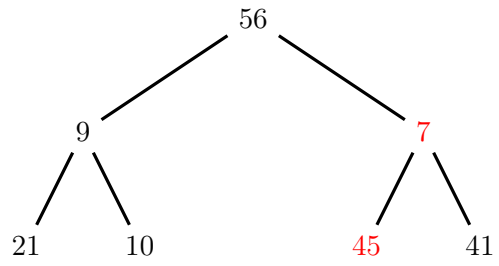


Figure 37: Swap 7 and 45

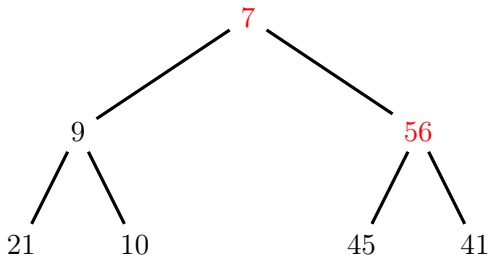


Figure 38: Swap 7 and 56

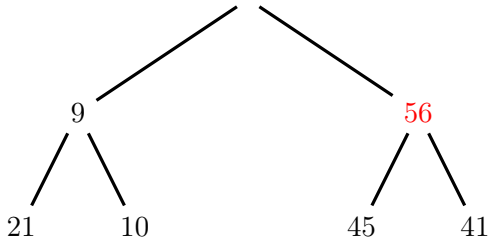


Figure 39: Remove 7

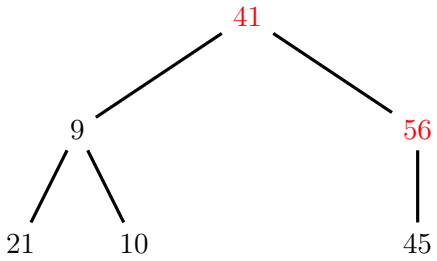


Figure 40: Move 41 to root

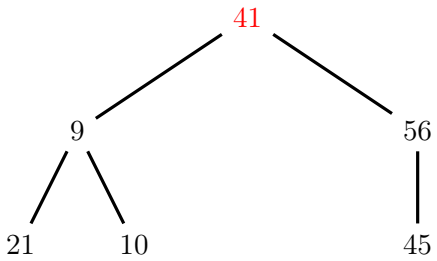


Figure 41: Move 41 to root

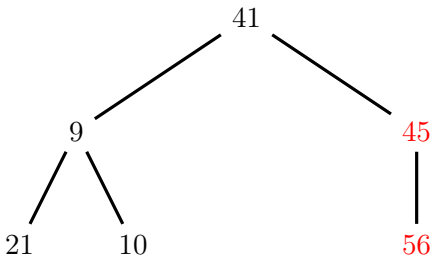


Figure 42: Swap 45 and 56

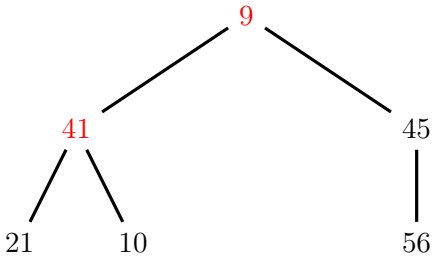


Figure 43: Swap 9 and 41

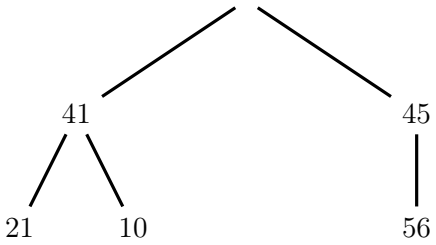


Figure 44: Remove 9

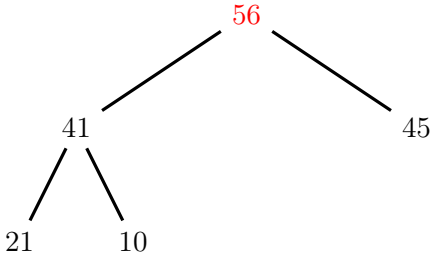


Figure 45: Move 56 to root

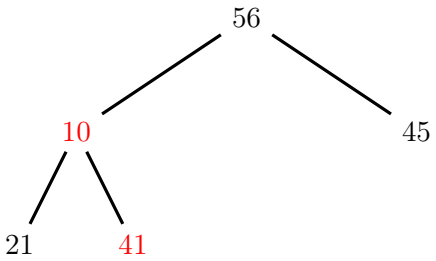


Figure 46: Swap 41 and 10

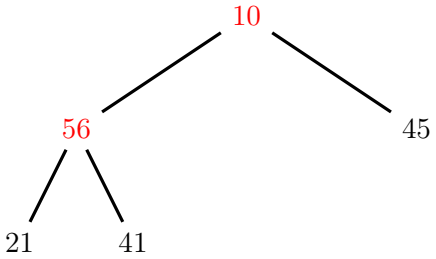


Figure 47: Swap 56 and 10

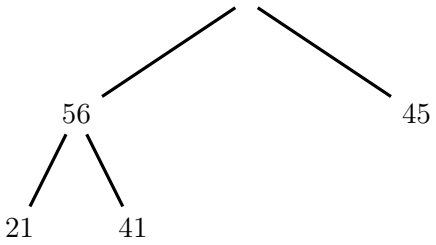


Figure 48: Remove 10

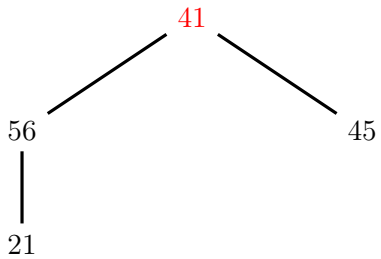


Figure 49: Move 41 to root

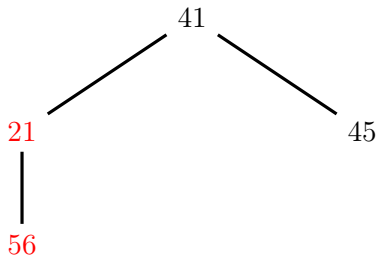


Figure 50: Swap 56 and 21

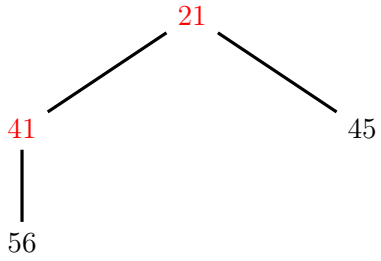


Figure 51: Swap 41 and 21

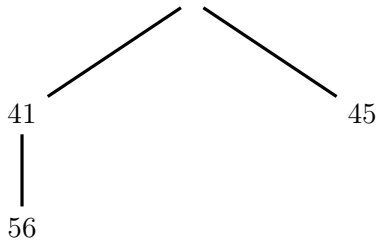


Figure 52: Remove 21

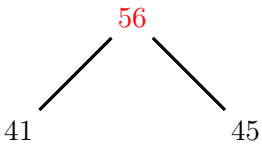


Figure 53: Move 56 to root

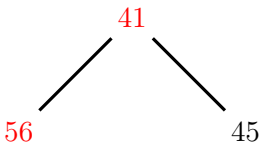


Figure 54: Swap 41 and 56

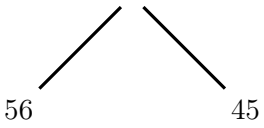


Figure 55: Remove 41

Figure 56: Move 45 to root

Figure 57: Remove 45

Figure 58: Move 56 to root

3	7	9	10	21	41	45	56

Figure 59: Sorted list after Heap Sort

Week 17

Learning Objectives

- Explore more advanced algorithmic techniques, including recursion and sorting algorithms.
- Consider merging lists, merge sorts and how they work.
- Understand and apply algorithms in the context of Shapley proofs and stable matching.

Essential Reading

- Rosen, K.H. Discrete mathematics and its applications. (New York: McGraw-Hill, 2012) 7th edition, Chapter 5.1, pp.307–327.

9.101 Recursion

A simple example

Here's the pseudocode for a function called `LessThan`

```
1 def LessThan(n):
2   Print n
3   If n-2 > 0:
4     LessThan(n-1)
5     LessThan(n-2)
```

What's the output when the input if 4? From line 2, we print 4. Then on line 3, we check if `n-2` is greater than 0, which it is; therefore we call `LessThan(n-1)` which evaluates to `LessThan(3)`.

Following we print 3, check if `n-2` is greater than 0, then call `LessThan(n-1)` which evaluates to `LessThan(2)`. Then we print 2, check if `n-2` is greater than 0. This time it isn't, then we can call exit our original function and call `LessThan(n-2)` where `n` is 3. We print `1` and exit this function. Now `n` is `4` and we call `LessThan(n-2)` which will print `2` and exit the function.

Our complete output is `4 3 2 1 2`.

Euclid's Algorithm: Greatest Common Divisor

Given two non-zero integers, `a` and `b`, find the greatest integer that divides `a` and `b` without leaving a remainder.

This algorithm relies on an important fact:

$$\text{GCD}(a, b) = \text{GCD}(b, r) = \text{GCD}(a, r)$$

The algorithm has the following steps:

1. assume $a \geq b$
2. divide a by b, the remainder is r
    a) By using a division once, the arguments to `GCD()` have decreased
    b) $b \leq a$
    c) $r < b$
3. repeat this process until $r = 0$, b is the final GCD.

Example: compute `GCD(84, 30)`

1. $84 \div 30 = 2, r = 24$
2. $30 \div 24 = 1, r = 6$
3. $24 \div 6 = 4, r = 0$

Now that the remainder `r` is 0, we can say that `b` is the `GCD(84,30)` which is 6.

Euclid's Algorithm: Pseudocode

The first version of the algorithm is iterative. Below we can find its pseudocode:

```
def GCD(a, b):
  while b != 0:
    t = b
    b = a mod b
    a = t
  return a
```

And a javascript implementation follows:

```
function GCD(a, b) {
  while (b !== 0) {
    const t = b
    b = a % b
    a = t
```

```
  }

  return a
}
```

The second version of the algorithm is recursive. Below we can find its pseudocode:

```
def GCD(a, b):
  if b = 0:
    return a
  return GCD(b, a mod b)
```

And a javascript implementation follows:

```
function GCD(a, b) {
  if (b === 0)
    return a
  return GCD(b, a % b)
}
```

9.103 Quick sort

Quicksort is a sorting algorithm that relies on recursion. The following pictures depict how it works:

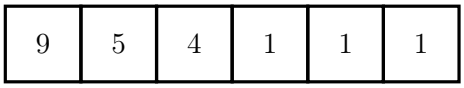


Figure 60: Unsorted array

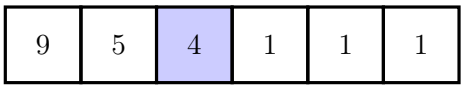


Figure 61: Pivot

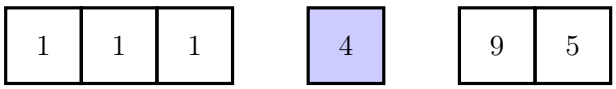


Figure 62: Sub-vectors

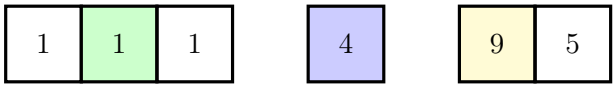


Figure 63: New pivots

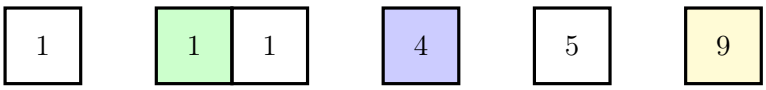


Figure 64: Split into smaller vectors

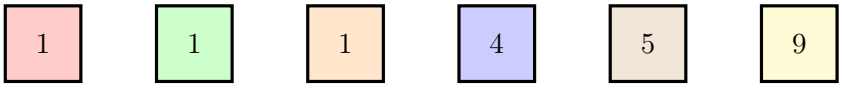


Figure 65: More pivots

Below, we can see the pseudocode for Quicksort. One thing to note is that since Quicksort is implemented recursively, we need a base case. That base case is a list with a single item, which is already sorted.

```
def QuickSort(List):
  If List has one item:
    Return List
  Else:
    Pivot = the middle entry of the List
    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
```

9.201 Merging lists

Merge Sort is another recursive sorting algorithm. The following sequence of pictures details the process or sorting using merge sort.

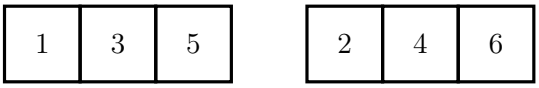


Figure 66: Initial state

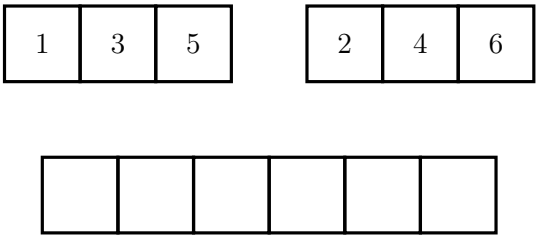


Figure 67: Create empty vector

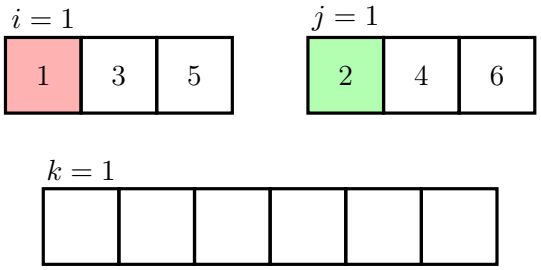


Figure 68: Compare elements

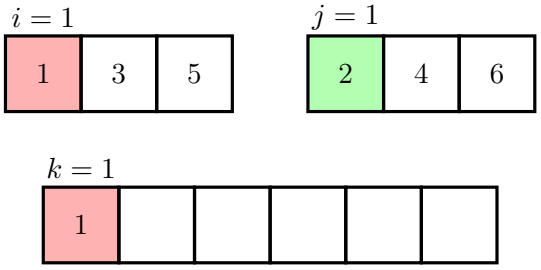


Figure 69: Move smallest to new vector

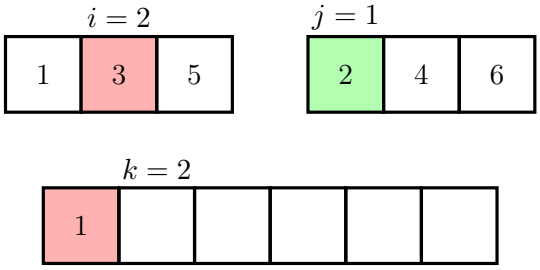


Figure 70: Increment i and k

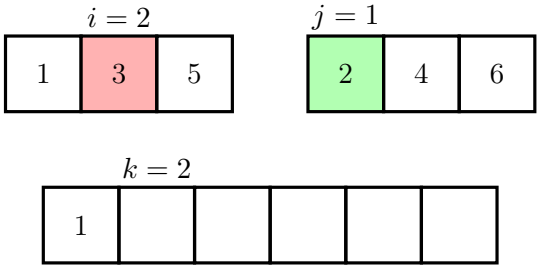


Figure 71: Compare elements

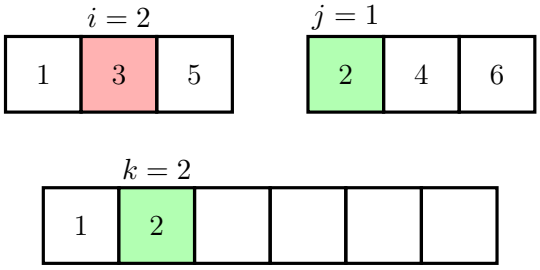


Figure 72: Move smallest to new vector

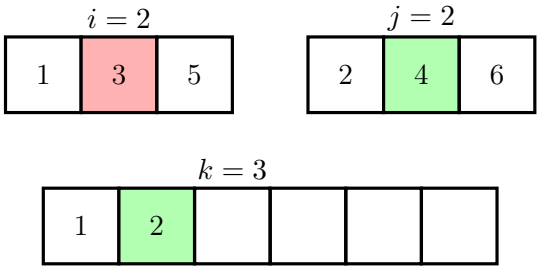


Figure 73: Increment j and k

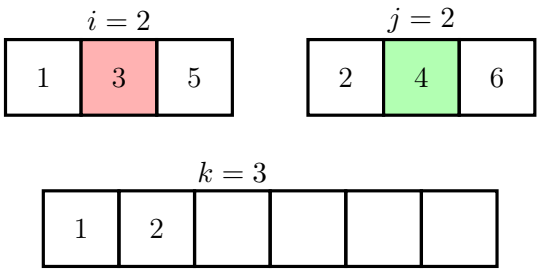


Figure 74: Compare elements

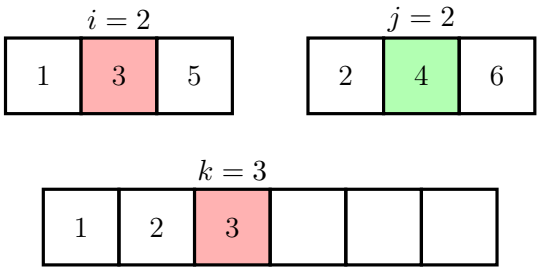


Figure 75: Move smallest to new vector

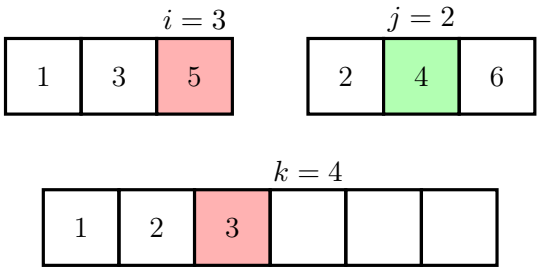


Figure 76: Increment i and k

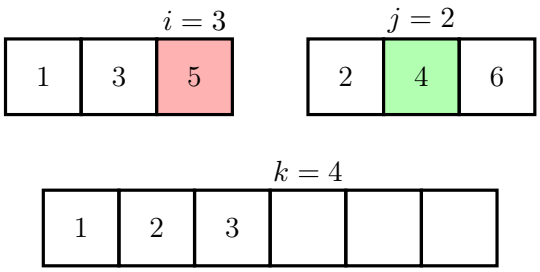


Figure 77: Compare elements

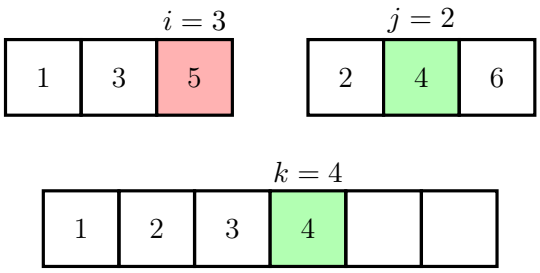


Figure 78: Move smallest to new vector

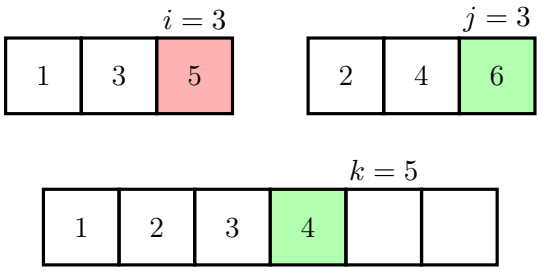


Figure 79: Increment j and k

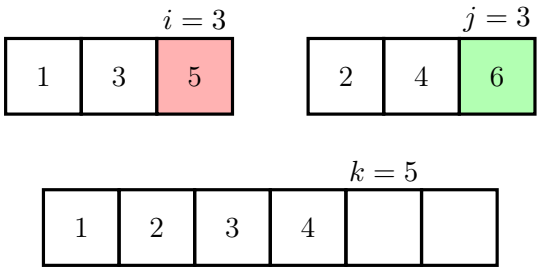


Figure 80: Compare elements

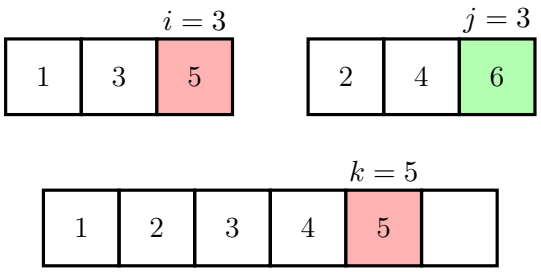


Figure 81: Move smallest to new vector

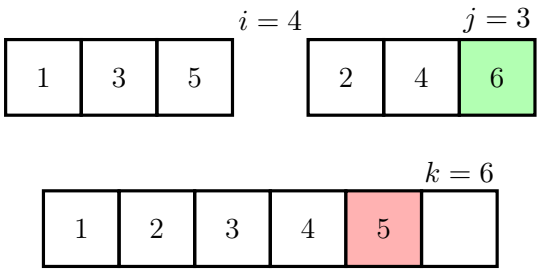


Figure 82: Increment i and k

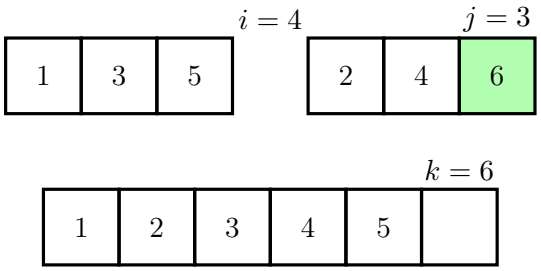


Figure 83: Only one element left

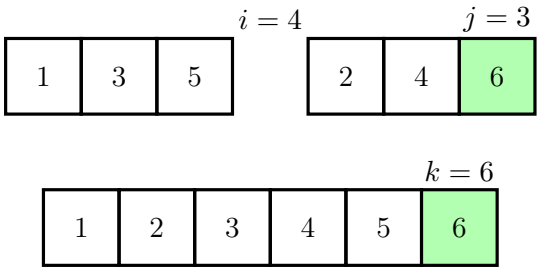


Figure 84: Move it to the new vector

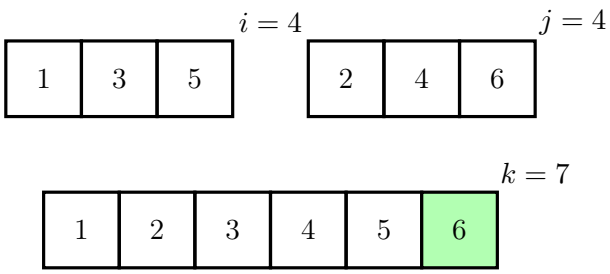


Figure 85: Increment j and k

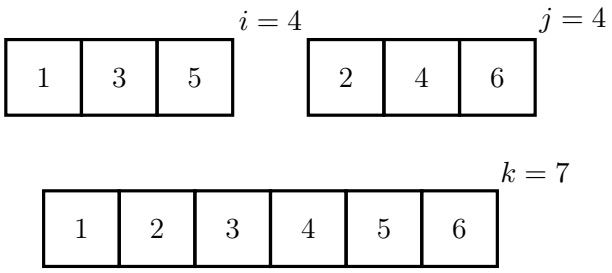


Figure 86: Algorithm terminated

How many comparisons?

- Every time two entires are compared:
  - one arrow moved to the right
  - only one space to the right
  - arrows never move to the left
- How many spaces can each arrow move?
  - the size of the list
  - In total: sum of list sizes

Merge sort pseudocode

```
1  def Merge(A, B):
2    while (A and B have elements):
3      if (A[1] > B[1]):
4        Append B[1] to the end of C
5        Remove B[1] from B
6      else:
7        Append A[1] to the end of C
8        Remove A[1] from A
9    while (A has elements):
10     Append A[1] to the end of C
11     Remove A[1] from A
12   while (B has elements):
13     Append B[1] to the end of C
14     Remove B[1] from B
15   return C
```

9.203 Merge sort

When we get the list, we keep splitting it into sub-lists until we have several lists of a single element which are all sorted.

Only then do we start merging and putting elements in their correct places.

Merge sort: Pseudocode

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

Week 18

Learning Objectives

- Explore more advanced algorithmic techniques, including recursion and sorting algorithms.
- Consider merging lists, merge sorts and how they work.
- Understand and apply algorithms in the context of Shapley proofs and stable matching.

9.301 The Algorithm of Happiness

The Gale-Shapley Algorithm[1, 2, 3] is an algorithm for finding a solution to the Stable Matching Problem.

Problem Statement

There are n hospitals and n medical students. There are lists of preferences for the students and the hospitals. Pair students to hospitals so that the matching is **stable**.

A pair student s and hospital h is called unstable when:

- Student s prefers hospital h to the one assigned to him/her
- Hospital h prefers student s to the one assigned to it

A stable matching is a matching without any unstable pairs.

The input to our algorithm will be:

- List of hospitals
- List of students

And a few rules:

1. Number of students is equal to number of hospitals

[1] Numberphile Video on Stable Marriage Problem
[2] Gale-Shapley Paper
[3] Mathematical Details of Stable Marriage Problem

2. One student per hospital
3. Each hospital provides an ordered list of students
4. Each student provides an ordered list of hospitals

For example:

Mohammed	UCLH	Whittington	Royal Free
Elena	Whittington	UCLH	Royal Free
Sara	Whittington	UCLH	Royal Free

Whittington	Mohammed	Elena	Sara
UCLH	Elena	Mohammed	Sara
Royal Free	Mohammed	Elena	Sara

Definitions

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

A perfect match

From the previous example, a possible perfect match would be as shown on by the blue cells below:

Mohammed	UCLH	Whittington	Royal Free
Elena	Whittington	UCLH	Royal Free
Sara	Whittington	UCLH	Royal Free

Whittington	Mohammed	Elena	Sara
UCLH	Elena	Mohammed	Sara
Royal Free	Mohammed	Elena	Sara

However, Elena and UCLH form an unstable pair because Elena prefers UCLH over her current hospital and UCLH prefers Elena over its current student. The existence of an unstable pair, makes the matching unstable.

$$\mathcal{M} = \{(M, U), (E, R), (S, W)\}$$

Gale-Shapley Algorithm, 1962

1. Each **unmatched** hospital offers a place to a student on the top of its list
2. Students

   With one offer accept the offer

   With more than one offer accept top hospital that made them an offer
3. Repeat until all hospitals are matched

9.303 The Gale-Shapley algorithm: example and pseudocode

Pseudocode: Gale-Shapley

```
M = emptyset
while (there is unmatched h and it has not been rejected by all students):
  s = first student on h's list who has not rejected h
  if s is unmatched:
    Add (h,s) to M
  else if s prefers h to current h':
    Delete (h',s) from M
    Add (h,s) to M
  else:
    s rejects h
Return M
```

Proof of correctness

In order to show that this algorithm terminates with a stable matching, we first show that it, indeed, terminates; then we show that it terminates with a stable matching.

- Prove that it terminates

  Look at the lines of pseudocode. We can see a single loop that runs for as long as there are students to offer. A hospital never offers to the same student twice and the student who rejects a hospital is removed from the list.

  Therefore, each hospital of the n offers at most n students. The loop runs at most n^2 iterations.

- Prove that it returns a perfect matching

Look at the definition of perfect matching. Perfect matching is when each hospital appears in the match at least **and** at most once (i.e. exactly once).

- At most

  Every hospital makes an offer **only if** it's unmatched. Every hospital appears in the match **at most** once.

  Every student accepts the best offer and rejects the others. Every student appears in the match **at most** once.

- At least (by contradiction)

  Assuming there is a hospital h that is not in the matching, then there must be at least one student s who is unmatched. This student has never received an offer. This means that h never made an offer to this student s.

  If the algorithm has terminated, then h has gone through all its list and it's still unmatched, however s must be in h's list, this is a contradiction.

- Prove that the match is stable

  No unstable pair!

  Let's assume there is an unstable pair (h, s). This means $(h, s') \in \mathcal{M}$ and $(h', s) \in \mathcal{M}$. There are few cases:

  1. s never got an offer from h

     Hospitals make offers in rank-decreasing order, which means that the rank of s' is higher than s, so (h, s) is stable.

  2. h made an offer to s

     s must have rejected the offer. Students only reject offers when they have better offers, which means rank h' must be higher than h, so (h, s) is stable.

Week 19

Learning Objectives

- Explore the concepts of efficiency, average, best and worst.
- Explore the usage of bubble sorts and binary searches.
- Consider asymptotic complexity and Big O notation.
- Explore recursion complexity and master theorem in the context of large problems.
- Consider efficient methods for problem solving.

10.101 Efficiency: insertion sort (time complexity)

Algorithms are analysed in the **time** they use and the memory **space** the need for the computation.

We estimate three efficiency scenarions:

Worst case this is when input would result in long computation times. It serves as an upper bound for any input.

Average case this is the average time taken to complete a task. Taken over all inputs. Serves as a more realistic expectation.

Best case the time taken in optimal scenario. Serves as lower bound for any input.

Sequential search - ordered list on n items

Worst case if the item is at the end of the list. n comparisons

Average case $$\frac{1+2+\ldots+n}{n} = \frac{\frac{1}{2}\cdot n(n+1)}{n} = \frac{n+1}{2}$$

Best case if the item is at the beginning of the list. 1 comparison.

10.103 Efficiency: bubble sort and binary search

See Algorithms and Data Structures I notes.

10.201 Asymptotic complexity

We look at asymptotic complexity of algorithms.

When analysing efficiency we need to consider how much is required for a given input size but also how fast this time grows as we increase the input size.

Graphs

Constant parallel to the x axis

logarithmic slow growth

Linear growth at a constant rate

Quadratic grows with the square of the input size

Exponential very fast growth

Asymptotic complexity

Refers to the growth of the function when the input size n is very large. It serves as a simple way of estimating how slow the algorithm is.

When doing such analyses, we ignore the lower order terms. For example, both $f(n) = 2n^2 + 5n$ and $g(n) = 4n^2 + 7n$ have the same asymptotic behavior.

Order of asymptotic behavior

1. Constant
2. Logarithmic
3. Linear
4. Linearithmic
5. Quadratic
6. Cubic
7. Higher order Polynomial
8. Exponential

10.203 Big O notation

Let $f(n)$ and $g(n)$ be two functions of n. We say that $f(n)$ is $\mathcal{O}(g(n))$ if there are constants c and k, such that:

$$f(x) \leq cg(x), x > k$$

We read this as $f(n)$ is Big $\mathcal{O}$ of $g(n)$. We refer to c and k as the witnesses.

Week 20

Learning Objectives

- Explore the concepts of efficiency, average, best and worst.
- Explore the usage of bubble sorts and binary searches.
- Consider asymptotic complexity and Big O notation.
- Explore recursion complexity and master theorem in the context of large problems.
- Consider efficient methods for problem solving.

Essential Reading

- Chang, S. (ed) Data structures and algorithms. (New Jersey: World Scientific Publishing, 2003), Chapters 8 and 9, pp.161–200.
  - Direct Link

10.301 Recursion complexity

What is recursion?

When a function is described by its value for a smaller input.

- Fibonnaci Sequence: $F(n) = F(n-1) + F(n-2)$, $F(1) = 1$, $F(2) = 1$.

We want to write these relations in terms of n only. There a few techniques for achieving this.

- Guess then prove by induction
- Tree method
- Master Theorem

  The recurrence $T(n) = aT\left(\frac{n}{b}\right) + \mathcal{O}(n^d)$, $a \geq 1$, $b > 1$, $d \geq 0$

$$
\begin{aligned}
d < \log_b a &\implies T(n) = \mathcal{O}(n^{\log_b a}) \\
d = \log_b a &\implies T(n) = \mathcal{O}(n^d \log n) \\
d > \log_b a &\implies T(n) = \mathcal{O}(n^d)
\end{aligned}
$$

10.304 Efficiency: Quick sort, merge sort

Quick sort: Worst case

The worst case happens when the pivot is the largest or the smallest element, resulting in either the right list or the left list being empty.

The total amount of comparisons for a list of n elements will be:

$$\mathcal{O}(n^2)$$

Quick sort: Average case

For the average case, we must make an average of all possible inputs.

The average amount of comparions for a list of n elements will be:

$$\mathcal{O}(n \log n)$$

Quick sort, Best case

The best case happens when the pivot is the median of the list, which will divide the list into two smaller lists at each iteration.

This means that at each iteration, our problem gets smaller and smaller.

$$\mathcal{O}(n \log n)$$

Merge sort: Worst case

$$\mathcal{O}(n \log n)$$

Merge sort: Average case

$$\mathcal{O}(n \log n)$$

Merge sort: Best case

$$\mathcal{O}(n \log n)$$