



COMPUTER MATHEMATICS AND LOGIC

STEVE KOLLMANSBERGER

<http://www.kolls.net/cml>

South Puget Sound Community College

Computer Mathematics and Logic
Compiled on September 1, 2011.

Copyright ©2011 Steven J Kollmansberger
For information contact steve@kolls.net
Or visit <http://www.kolls.net/>



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Typeset in L^AT_EX.

Dedication

This work is dedicated to those who shaped my academic ambitions and provided personal encouragement along the way.

My Parents
Denise Sumner
Dr. Kent Jones, Ph.D.
Dr. Susan Mabry, Ph.D.
Dr. Michael Quinn, Ph.D.
Dr. Martin Erwig, Ph.D.
Lisa Kollmansberger

And to all my students who have encouraged my efforts to provide the best educational opportunities.

Preface

The breadth of topics in computer mathematics and logic have the potential to be among the most interesting computational subjects you will study. Take out your phone. Start typing a text message. How are the characters and words represented (Chapter 12)? How does the phone calculate how many more characters you can type in the message (Chapter 10)? Attach a photo. How is an image stored inside the phone (Chapter 13)? Send the message and image across the wireless network. How is the information sent quickly (Chapter 16) and how are errors prevented (Chapter 15)?

How is the phone's software designed (Chapters 6, 19)? Even with such a small processor, how were the designers able to make sure it would perform fast enough (Chapter 20)? Imagine the insides of the phone. What are the "chips" made of (Chapter 8) and how do they make the phone perform all its tasks (Chapter 17)?

This book offers to take you on a conceptual journey starting with only your knowledge of basic algebra, all the way through the techniques of designers and programmers, and to explain along the way how the miracle of modern computation is achieved.

This book was borne out of frustration with existing offerings in the computer mathematics and logic arena. Many books are written for upper division undergraduates or graduate students. Many books focus on limited subsets of the topics presented herein, or stray far and wide into unrelated topics. Few books are willing to take on a broad spectrum of computer math at an introductory level. Of books that do venture into such territory, many are painfully dated, discussing with gusto obsolete encodings and omitting modern standards.

This book is designed for the student with little to no computer programming or college math experience, but who has completed high school algebra. Each topic is presented with an emphasis on conceptual understanding; as a result, not all topics are explored in complete depth. Specific advanced texts on each topic should be consulted should the student wish to delve further beyond the basics. In order to remain within the basic algebra requirement, certain subtopics requiring more advanced math have been omitted. In all cases explanations emphasize practical and down-to-earth approaches rather than highly technical mathematical definitions.

Layout

Throughout this text, pages will be divided into two columns: the main column contains the primary text, and the sidebar contains highlights, placed into boxes.

Definitions are used to give a concise meaning of a term that is likely to appear again throughout the chapter or beyond. In all cases, short and practical definitions are preferred over detailed and precise technical mathematical definitions.

Certain mathematical properties or formulas may be shown using the math box in the sidebar. These formulas may not be essential to being able to work with the concepts described, but understanding them (and why they are true!) is likely to be a great benefit. You should work on discovering why these formulas are true instead of simply accepting them at face value.

In some chapters, computer software has been employed to assist in the creation of figures. In other cases, various websites or software is recommended to help work through certain types of problems. Links for recommended websites and software will appear in a box with the computer icon. The computer icon may also be used to indicate computer or programmer related notes or limitations.

■ **Example 0.1** • Many concepts will be illustrated with examples. To help avoid confusion, examples of more than few sentences in length will be marked and numbered. ■

Definitions are shown with the dictionary icon. All definitions also appear in the glossary in the back.

Insights give helping hints for problem solving, or possible pitfalls to watch out for.

$\sum_{n=1}^{\infty} \frac{1}{n^2}$ If n is negative, and p is positive, then $n < p$

 The author's website is <http://www.kolls.net>

CONTENTS

I Sets and Counting

1 Sets	2
2 Counting	9
3 Venn Diagrams	19
4 Simplifying Set Expressions	27

II Boolean Logic

5 Logical Operators and Truth Tables . . .	37
6 Manipulating Logical Expressions . . .	48
7 Decision Tables	62
8 Logic Circuits	73

III Numbers and Letters

9 Number Systems	90
10 Integer Numbers	100
11 Floating Point Numbers	112
12 Unicode and ASCII	123

IV Data Representation

13 Images and Color	131
14 Bitwise Operations and Masking . . .	145
15 Error Correcting Codes	155
16 Compression	168

V Algorithms

17 Digital Logic	170
18 Finite Automata	196
19 Flowcharts	197
20 Analysis of Algorithms	223
21 Expression Trees	238

VI Supporting Material

A Solutions	240
B Bibliography	373
C Glossary	377

PART I

Sets and Counting

1 Sets	2	3 Venn Diagrams	19
1.1 Set Notation	3	3.1 Set Operations	20
1.2 Set Operations	4	3.2 Other Set Operations . . .	22
1.3 Exercises	8	3.3 Creating a Venn Diagram .	22
2 Counting	9	3.4 Higher Order Venn Dia-grams	24
2.1 Additive and Multiplica-tive Counting	9	3.5 Exercises	26
2.2 Preliminaries	10	4 Simplifying Set Expressions	27
2.3 Two Key Questions	12	4.1 Algebra Laws of Sets . . .	27
2.4 Sample Applications	12	4.2 Explanation of Laws . . .	28
2.5 Explanation of Formulas .	14	4.3 Applying Algebra Laws .	30
2.6 Partition Rule	17	4.4 Exercises	35
2.7 Exercises	18		

Sets



Set: an unordered collection of items without duplicates.



Empty set: the set containing no elements.



Universe: the set containing all possible items under consideration.



Finite Set: a set containing a limited (although possibly very large) number of elements.



Infinite Set: a set containing an unlimited number of elements, usually defined mathematically.

A set is an unordered collection of items without duplicates. When we say “item”, we refer to any type of thing: a set could consist of numbers, of letters, of animals, of planets, of people, of buildings, and so on. For example, all the people who have read this book form a set. All positive numbers also form a set. You could define the set of all elephants in India. How about the set of pink flying elephants? This is also fine, but it is likely that this set contains no elements. A set which contains no elements is referred to as the empty set.

When referring to sets it is helpful to always keep in mind the universe for those sets. The universe is usually the largest set of items of the same type as the sets in question. In some cases, the universe may be implicit. In other cases, it should be defined directly. For example, in the set of all people who have read this book, the universe is likely to be all people. With the set of all elephants in India, is the universe all elephants in the world? Or is it all animals in India? Or... In such a case, and in most cases, it is best to define the universe explicitly so there is no possibility of confusion.

Sets may be finite (meaning they have a limited number of elements) or infinite. Finite sets would include all people on planet Earth. Infinite sets are usually numerical and include sets like all positive numbers. Even sets which appear “small” could be infinite: for example, the set of all reals between 1 and 2 is infinite.

1.1 Set Notation

To facilitate discussion, a standard notation is usually employed to describe sets. A set is often assigned a name, usually just a single uppercase letter. The enumeration (list of elements) of the set itself is wrapped in curly brackets and the elements are listed out, separated by commas. For example, consider the set of all whole numbers between 1 and 10, inclusive. If we called this set A , we could write it:

$$A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

Keep in mind that sets are unordered, which means that the sequence of elements written has no meaning. If we define B as follows:

$$B = \{10, 1, 9, 2, 8, 3, 7, 4, 6, 5\}$$

Then $A = B$ (or, A is equivalent to B) because both sets contain the same elements. Whenever two sets contain the same elements, they are said to be equivalent.

The two sets defined above, A and B are both finite sets described using enumeration. Infinite sets can be described using a conceptual notation. For example, consider the set of all positive integers:

$$C = \{x : x \text{ is a positive integer}\}$$

Here x is a variable representing a single element of the set, and the range of x is described in words.

Here are some other examples of sets. The set of primary colors could be defined as $P = \{\text{red}, \text{green}, \text{blue}\}$. The set of adult males in Canada could be defined as $M = \{q : q \text{ is an adult male in Canada}\}$. Note the use of the conceptual notation, since the specific names of all adult males in Canada is not immediately known (and changes). The label q is simply a variable. The variable is usually x , but q is used here to demonstrate that any variable is possible.

In addition, special notation exists for the empty set and the universe. The empty set is designated with \emptyset or sometimes $\{\}$, and the universe is designated with

Unlike lists, sets have no ordering. The sequence that elements are written in is coincidental and has no meaning.



Equivalent: two sets which contain the same elements are equivalent.

\mathcal{U} .



Cardinality: the number of elements in a set.

The number of elements in a set is called the cardinality of that set, and shown with vertical lines on either side of the set. For example, $|A| = 10$, given the value of A above.

Membership in a set is denoted by, for example, $3 \in A$, which claims the number 3 is in the set A (defined above). A claim of non-membership is made by slashing the operator, as in $26 \notin B$. These operators can be used with the conceptual notation to define a set. For example $D = \{x : x \in C, x \notin A\}$. This set D then contains all positive integers greater than 10.

1.2 Set Operations



Union: the set of all elements that appear in either or both of the original sets.



Intersection: the set of all elements that appear in both of the original sets.



Complement: the set of all the elements in the universe that do not appear in the original set.

Sets can be manipulated using three main operators: union, intersection, and complement. The union of two sets is the set of all elements that appear in either or both of the original sets. Union is defined with the symbol \cup . The intersection of two sets is the set of all elements that appear in both of the original sets. Intersection is defined with the symbol \cap . The complement of a set is the set of all the elements in the universe that do not appear in the original set. The complement only has meaning if the universe is known or somehow described. Complement is defined with the ' or sometimes c .

■ **Example 1.1** • Assume we define the sets $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$. In that case, the union of the two sets, represented as $A \cup B$, contains the five elements which appear in either or both sets, namely $\{1, 2, 3, 4, 5\}$. Remember that order is not important, so it's also true that $A \cup B = \{5, 4, 1, 3, 2\}$.

Continuing with the previous definition of A and B , the intersection of the two sets, represented as $A \cap B$, contains the individual element which appears in both sets, namely $\{3\}$. In the event that the sets did not share any elements in common, the intersection would be the empty set. For example, $\{1, 2, 3\} \cap \{5, 6, 7\} = \emptyset$. In this



case, the sets are referred to as disjoint.

In order to determine the complement of some set, we must first know what the universe is. For this example, we'll define the universe as $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. In that case, $A' = \{4, 5, 6, 7, 8, 9, 10\}$, that is, all elements in the universe which do not appear in A . ■

Set operations can be combined into set statements or expressions using a standard order of operations. Parentheses are evaluated first, then complement, then intersection, and finally union.

■ **Example 1.2** • Using the definitions of A , B , and \mathcal{U} provided previously, consider the expression $A \cap B'$.

1. $A \cap B'$
2. $\{1, 2, 3\} \cap \{3, 4, 5\}'$ (insert set definitions)
3. $\{1, 2, 3\} \cap \{1, 2, 6, 7, 8, 9, 10\}$ (apply complement)
4. $\{1, 2\}$ (apply intersection)

Therefore, $A \cap B' = \{1, 2\}$. ■

■ **Example 1.3** • Consider a more complex example: $A \cup B \cap A'$. Intersection has higher precedence than union, so the intersection will be evaluated before the union (but after the complement).

1. $A \cup B \cap A'$
2. $\{1, 2, 3\} \cup \{3, 4, 5\} \cap \{1, 2, 3\}'$ (insert set definitions)
3. $\{1, 2, 3\} \cup \{3, 4, 5\} \cap \{4, 5, 6, 7, 8, 9, 10\}$ (apply complement)
4. $\{1, 2, 3\} \cup \{4, 5\}$ (apply intersection)
5. $\{1, 2, 3, 4, 5\}$ (apply union)



These operations apply to infinite sets as well.



Be sure not to place duplicate elements in the result of a union operator. Sets don't contain duplicates!



Disjoint: two sets that do not share any elements in common.



$$|A \cup B| = |A| + |B| - |A \cap B|$$



A' may be read as *A prime*



■ **Example 1.4** • Let $\mathcal{U} = \{x : x \text{ is an integer}\}$, $A = \{x : x \text{ is a positive integer}\}$, and $B = \{1, 2, 3, 4, 5\}$. What set is described by $A' \cap B'$?

1. $A' \cap B'$
2. $\{x : x \text{ is a positive integer}\}' \cap \{1, 2, 3, 4, 5\}'$ (insert set definitions)
3. $\{x : x \text{ is a negative integer, or zero }\} \cap \{x : x \text{ is an integer less than } 1 \text{ or greater than } 5\}$ (apply complements - notice that B is finite but B' is infinite)
4. $\{x : x \text{ is a negative integer, or zero }\}$ (apply intersection - notice that we have to reason about how these sets may overlap, specifically that the “negative” part of the first set overlaps with the “less than 1” part of the second)

Therefore $A' \cap B' = \{x : x \text{ is a negative integer, or zero }\}$. ■

 When dealing with described numeric sets, consider drawing a number line to help reason about what the union, intersection, or complement will be.

 **Subset:** all of the elements of a set are contained within another set.

$\sum_{n=1}^{\infty} \frac{1}{n^2}$ If $A \subseteq B$, then $|A| \leq |B|$

$\sum_{n=1}^{\infty} \frac{1}{n^2}$ If $A \subseteq B$ and $B \subseteq A$, then $A = B$

 **Proper Subset:** all of the elements of a set are contained within another set, and the other set also has at least one additional element.

$\sum_{n=1}^{\infty} \frac{1}{n^2}$ If $A \subset B$, then $|A| < |B|$

$\sum_{n=1}^{\infty} \frac{1}{n^2}$ If $A \subset B$, then $A \subseteq B$

In addition to equivalence, described earlier, sets can also be related through the concept of subset. A set is a subset of another set if all of the elements in the first set appear in the second. The subset is shown as $A \subseteq B$, indicating that all elements in A appear in B . It is also possible that $A = B$.

A special form of subset is the proper subset, which strengthens the claim of subset. $A \subset B$ claims that not only is every element in A also in B , but there is some element in B that does not appear in A . If $A \subset B$, then it is not possible that $A = B$.

If there is an element in A which does not appear in B , then we can say $A \not\subseteq B$.

■ **Example 1.5** • For example, let's assume that $A = \{1, 2, 3\}$, $B = \{3, 4, 5\}$, $C = \{1, 5, 7\}$ and $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Determine if $A \cup (B \cap C) \subset C$.

1. $A \cup (B \cap C) \subset C$
2. $\{1, 2, 3\} \cup (\{3, 4, 5\} \cap \{1, 5, 7\}) \subset \{1, 5, 7\}$ (insert set definitions)

3. $\{1, 2, 3\} \cup (\{5\}) \subset \{1, 5, 7\}$ (parentheses first - perform intersection)
4. $\{1, 2, 3, 5\} \subset \{1, 5, 7\}$ (perform union)
5. $\{1, 2, 3, 5\} \subset \{1, 5, 7\}$ (check if all elements in the left set appear in the right set; plus at least one more element in the right set)

We do not find all elements from the first set in the second set, therefore, $A \cup (B \cap C) \not\subset C$. ■



1.3 Exercises

Solutions to these exercises can be found in Appendix A.1 on page 241.

Assume $A = \{1, 2, 3\}$, $B = \{2, 3, 4\}$, $C = \{4, 5, 6\}$, $D = \{1, 3, 5\}$, $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

1. *Problem:* Find the set described by $A \cup (B \cap D')'$.
2. *Problem:* Find the set described by $A \cup B \cap D$.
3. *Problem:* Determine if $A \cap B \subset B \cup D$.
4. *Problem:* Find $|A \cap C|$.

Assume $M = \{x : x \text{ is a math student}\}$, $C = \{x : x \text{ is a CIS student}\}$, and $V = \{x : x \text{ plays videogames}\}$. Let $\mathcal{U} = \{x : x \text{ is a student at the college}\}$. Note that all sets must be subsets of the universe, so in this case the

set V implicitly is limited to only those students at the college who play videogames; not videogame players outside the college.

5. *Problem:* Write a set expression which gives the set of all CIS students who play videogames.
6. *Problem:* Write a set expression which gives the set of all students who play videogames and are either math or CIS students, or both.
7. *Problem:* Write a set statement which indicates that all CIS students play videogames.
8. *Problem:* Write a set statement which indicates that some math students don't play videogames.



Chapter 2

Counting

Permutations and combinations describe a set of mathematical rules used for counting the number of possible arrangements of a set of items. Knowing how many possibilities we may need to deal with is important to ensure that a computer program has sufficient capacity to handle whatever may possibly come its way.

2.1 Additive and Multiplicative Counting

If we are choosing just one item from two sets, A and B , then the number of possible choices is $|A| + |B|$.

For example, let's say $A = \{\text{salad, soup}\}$ and $B = \{\text{ham, eggs, pancakes}\}$. If we are to choose *one* of these, then there are a total of five possible choices. Specifically, $C = \{\text{salad, soup, ham, eggs, pancakes}\}$.

On the other hand, let's say that instead of choosing just one item, we chose one item from each set. In other words, if we have n sets, then we are choosing n items, one from each set. Using the same sets as above, we might choose salad and ham; alternatively, we might choose soup and pancakes. In this case, there are a total of six possible choices, defined by $|A| * |B|$. Specifically, $C = \{\{\text{salad, ham}\}, \{\text{salad, eggs}\}, \{\text{salad, pancakes}\}, \{\text{soup, ham}\}, \{\text{soup, eggs}\}, \{\text{soup, pancakes}\}\}$.

Additive Counting Rule:
When selecting just one item from several sets, the total number of possibilities is the sum of the cardinalities of the sets.

 **Multiplicative Counting Rule:** When selecting one item from each of several sets, the total number of possibilities is the product of the cardinalities of the sets.

eggs}, {salad, pancakes}, {soup, ham}, {soup, eggs}, {soup, pancakes}}.

These two rules are the foundation for more complicated techniques of counting.

2.2 Preliminaries

 **Factorial:** the product of integers from 1 up thru some given number, indicated by the exclamation point.

$$\sum_{n=1}^{\infty} \frac{1}{n!} \quad \text{Let } 0! = 1, \text{ and then } n! = (n-1)! * n$$

 **Binomial Coefficient:** Formally, coefficient of the x^r term in the polynomial expansion of $(1+x)^n$. Practically, the number of ways to select r items from a set of n .

$$\sum_{n=1}^{\infty} \binom{n}{n-r} = \binom{n}{r}$$

 Many calculators support the binomial coefficient with a button labeled nCr , short for n choose r .

It is convenient to define two mathematical functions to assist us in further definitions. The factorial of a positive integer n is defined as the multiplication of all integers starting at 1 and going up to and including n . The factorial is indicated with the ! symbol. For example, $5! = 1 * 2 * 3 * 4 * 5 = 120$.

Let's say that $5! = 120$ has been found above. What is $7!$? We could define $7! = 7 * 6 * 5!$ (because $5!$ provides the multiplication of all remaining numbers). Thus $7! = 7 * 6 * 120 = 5040$. When solving multiple factorial problems, this shortcut can save some time.

The factorial definition can be used in the definition of the binomial coefficient. The binomial coefficient, written as $\binom{n}{r}$, defines the number of ways to select r items from a set of n (more on this later). The binomial coefficient is defined in terms of factorial:

$$\binom{n}{r} = \frac{n!}{r! * (n-r)!}$$

Another way to determine the binomial coefficient is to use Pascal's Triangle. Pascal's Triangle is, in fact, based on the polynomial coefficients but these values can be determined row-by-row in the triangle. The top row contains just the value 1. The second row contains the values 1 and 1. Then, for each subsequent row, place 1s on the outside and fill the middle by adding the two values above. This is easiest to understand through a diagram:



$n = 0:$	1
$n = 1:$	1 1
$n = 2:$	1 2 1
$n = 3:$	1 3 3 1
$n = 4:$	1 4 6 4 1
$n = 5:$	1 5 10 10 5 1

Notice the 2 in the third row is the sum of the two 1s above it (diagonally). Likewise, the 3 in the fourth row is the sum of 1 + 2, the two numbers immediately above it. This property continues throughout the triangle. The 10 in the sixth row is defined as 4 + 6, the sum of the two numbers above it.

Why is Pascal's Triangle interesting? For any binomial coefficient, $\binom{n}{r}$, we can choose the n^{th} row (with the first row being $n = 0$), and then from left to right, choose the r^{th} value (with the first value being $r = 0$). This is the result of the binomial coefficient.

■ **Example 2.1** • For example, using the triangle above, let's solve $\binom{4}{2}$. First we choose the fifth row (where $n = 4$), and then count over to the third value (the first value would be $r = 0$, and so on). This value is 6. Therefore, $\binom{4}{2} = 6$.

We can check this work by using the factorial definition: $\frac{n!}{r! * (n - r)!} = \frac{4!}{2! * (4 - 2)!} = \frac{1 * 2 * 3 * 4}{1 * 2 * 1 * 2} = \frac{24}{4} = 6$. ■

Using these two tools of factorial and binomial coefficient, we are ready to define four key permutation and combination counting formulas.

Any row in Pascal's Triangle can be calculated directly by expanding $(a+b)^n$ where n is the row number. For example, if $n = 5$ then $(a+b)^5 = a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$. The coefficients here are, in order 1, 5, 10, 10, 5, 1 - the same as that row in Pascal's Triangle.



2.3 Two Key Questions

In order to solve a counting problem which falls into the permutation/combinatorial category (that is, selecting or choosing some items from a set of items), two questions must be answered:

First, is order important? For example, if I am choosing numbers for a combination lock, then 4-2-1 is different from 2-1-4. In this case, order is important. On the other hand, consider the case where I want to elect 4 representatives from a group of 100 people. In that case, each representative is equal and so order is not important.

If we want to select more items than actually exist in the original set, repetition must be allowed and used. For example, if I order 10 appetizers from a menu of 8 appetizers, there must be some repetition.



A fun and easy calculator with permutations and combinations can be found at <http://www.mathsisfun.com/combinatorics/combinations-permutations-calculator.html>



Permutation: A selection of items from a set where the order of selection is important.



Combination: A selection of items from a set where the order of selection is *not* important.



Second, is repetition allowed? For example, if I am selecting four appetizers from a menu of ten, I could choose the same appetizer twice or more. In this case, repetition is allowed. On the other hand, to use the same representatives example as above, the 4 representatives must all be distinct individuals - one person could not serve as three of the representatives. In that case, repetition is not allowed.

If order is important, then one of the *permutation* formulas is used. If order is not important, then one of the *combination* formulas is used.

Both permutations and combinations can appear either with repetition, or without. This forms four possibilities. Assuming that we have a set of n items and we are choosing r of them, here are the formulas:

	Repetition	No Repetition
Permutation	n^r	$\frac{n!}{(n-r)!}$
Combination	$\binom{n+r-1}{r}$	$\binom{n}{r}$

2.4 Sample Applications

It can be difficult to determine when to use which formula (much more difficult than simply applying the

formula). To assist in this determination, here are four example situations, one corresponding to each formula.

■ **Example 2.2** • In the case of permutations with repetition, consider the case of determining how many different license plates are possible. A license plate consists of a sequence of symbols. The order of the symbols matters (so ABC is different from CBA) and it is allowable to reuse a symbol (so the sequence AA11 could appear on a plate).

Thus, the number of license plates could be counted using permutations with repetition. If a certain kind of license plate had six symbols, and each symbol could be an uppercase letter or digit, then the number of possible plates would be found with 36^6 . ■

■ **Example 2.3** • In the case of permutations without repetition, consider the case of a karaoke contest. In the contest, each performer sings once (so no repetition), but performers must be ordered. It is a different sequence if performer A goes first or last, and so on.

Thus, the number of ways to sequence a karaoke contest could be counted using permutations without repetition. If a certain contest had 13 contestants (and all 13 were to sing), then the number of possible sequences would be found with $\frac{13!}{(13 - 13)!}$. ■

■ **Example 2.4** • In the case of combinations without repetition, consider the case of the selection of representatives from a group. A certain group of people wish to elect (or have selected) a subset of representatives from among them. In this case, each spot has to be filled by a different person (if I ask for five representatives, you can't send one person and claim that person fills all five spots). Likewise, there is no significance of ordering; the group is traveling as a set (no duplicates, no ordering). In any case where the selection is to produce a set, combinations without repetition is probably appropriate.

Thus, the number of ways to select a set of representatives could be counted using combinations without



repetition. If a certain group had 120 people, and 7 representatives were selected, then the number of possible representative sets would be found with $\binom{120}{7}$. ■

■ **Example 2.5** • Finally, in the case of combinations with repetition, consider the case of a raffle of identical items. Assuming no limit on the distribution of raffle tickets, it is possible for one person to win all the prizes in a raffle. In this case, the repetition comes from selecting the winner; the same winner may be selected multiple times (if they hold multiple tickets). Likewise, since all prizes are identical, there is no concept of ordering: the first winner and the last winner receive exactly the same prize.

Thus, the number of ways to select the winners of a raffle with identical prizes and no limit on tickets could be counted using combinations with repetition. If there were a group of 15 players (each with “unlimited” tickets), and 3 prizes, then the number of ways the prizes could be distributed would be found with $\binom{15 + 3 - 1}{3}$. ■

2.5 Explanation of Formulas

In order to calculate permutations and combinations, the previously given four formulas, together with the definitions of factorial and binomial coefficient, are sufficient (as long as we are able to determine correctly whether or not order is important and whether or not repetition is allowed).

However, discovering a little more about why these formulas work may be interesting.

Permutations with repetition, defined by n^r , indicates that we are choosing out of n items r times, and that we may repeat some items. In this case, for the first item we can choose one of n possibilities. For the second item, we can, again, choose from n possibilities, and so on. In general, independent choices are multi-



plied together to find the total number of choices possible. Therefore, we are multiplying $n * n * n * n \dots$ up to r times. This is based on the multiplicative counting rule. Exponent notation gives a shorthand for this in n^r .

How about permutations without repetition, defined by $\frac{n!}{(n-r)!}$? If I want to choose r items out of n items, without repetition, then every time I make a choice I have one less item available. For the first item, there are n possibilities. For the second, there are $n - 1$ possibilities. For the third, $n - 2$ possibilities, and so on. The definition of factorial, $n!$, starts out this way: $n! = n * (n - 1) * (n - 2) * (n - 3) * \dots * 3 * 2 * 1$. However, we only want to have the first r terms of the factorial, not all n terms.

In order to eliminate the later factorial terms that are undesirable, they are canceled using division by $(n - r)!$. The term $(n - r)!$ will start at the first unneeded term and proceed down to 1.

■ **Example 2.6** • This is best seen by use of an example. Let's assume we have $n = 10$ and $r = 3$. That is, we have 10 items and wish to choose 3 of them without repetition. In that case, using the multiplicative counting rule, there are $10 * 9 * 8$ possibilities. The value $10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1$. So if we could cancel out the $7 * 6 * 5 * 4 * 3 * 2 * 1$ part, that answer would be correct (it would give us the $10 * 9 * 8$ we're looking for). Notice the value we wish to cancel is simply $7!$, the same as $(10 - 3)!$.

$$\frac{n!}{(n-r)!} = \frac{10!}{(10-3)!} = \\ \frac{10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1}{7 * 6 * 5 * 4 * 3 * 2 * 1} = \\ 10 * 9 * 8 = 720$$

Combinations without repetition, expressed with the binomial coefficient $\binom{n}{r}$ can be explained with a similar reduction. First, keep in mind that the binomial coefficient expands to $\frac{n!}{r! * (n - r)!}$. With the exception

Permutations with repetition is commonly used in computer programming to determine how large of a number can be stored in a certain amount of space, or how many items can be addressed with a certain address size. See the exercises at the end of this chapter for examples.



Disallowing repetition will never result in more choices.





If repetition is held constant, the number of possible combinations is always less than or equal to the number of permutations.

of the $r!$ term on the bottom, this is the same as the permutations without repetition just shown.

For combinations (where order does not matter) we start with the formula for permutations (where order does matter) and eliminate those cases which are duplicated due to ordering. How many cases is that? Assume we are choosing three items ($r = 3$). In that case, if we choose items a, b , and c , then there are $3!$ possible orderings of these specific values (by the multiplicative counting rule). We want to reduce that down to one possible selection (as order does not matter).

This is why, if we take the permutations without repetition, we can remove ordering by dividing the result by $r!$ where r is the number of elements we selected.

Finally, consider combinations with repetition, represented by $\binom{n+r-1}{r}$. The best way to think about combinations with repetition is to imagine that an arbitrary ordering has been placed on the items to be selected. We then go down the list, saying either “take” or “next”. As long as we say “take”, we get one of those items (and stay on the same item in the list). When we say “next” we move to the next item. The sequence ends when all elements have been considered. In other words, there must be exactly $n - 1$ “next”s. The reason for $n - 1$ is that we start, by default, on the first item and so there is no “next” to arrive there. Further, because we are choosing r items, there must be exactly r “take”s.

Think of this as having $r + (n - 1)$ positions, and of these positions, r of them will be selected to be “take”s and the remaining $n - 1$ will be “next”s. The order in which we select the “take”s is not important – each position is unique, but if I say I’m going to take the first, and the third, that’s the same as saying I’m going to take the third, then the first.

We can represent the collection of “take”s and “next”s as a list of commands (the commands being “take” or “next”). The list of commands, consisting of $n - 1$ “next”s and r “take”s, is run in sequence. The absence of a “next” command causes two adjacent “take” com-



mands to select the same item.

In order to specify the contents of this list, we can start the list consisting entirely of “next”s, and then select which positions in the list to replace the “next” with a “take”. Repetition is not allowed because we are choosing positions in the command list, and each position has exactly one command. So, for example, if position 3 is chosen to hold a “take”, we would only select position 3 once.

We can then represent combinations with repetition using a modified formula and combinations without repetition: we are choosing, without repetition, r positions in an $r + (n - 1)$ command list to “take”. Plugging these values into the formula for combinations with repetition gives us $\binom{n + r - 1}{r}$.

2.6 Partition Rule

The partition rule allows a form of combination without repetitions when multiple groups (of different sizes) are used. The total size of all groups must equal the original number of elements; if not, a final “blank” group can be created. Assume n is the number of objects to choose from, and the group sizes are denoted by r_1, r_2, r_3, \dots where all these values sum to n .

In that case, the number of possible selections is represented by $\frac{n!}{r_1! * r_2! * r_3! * \dots}$.

■ **Example 2.7** • For example, assume we have 10 children (it’s a daycare or something). We want to assign three to clean up the yard, four to help paint the downstairs and three to wash the family car. In how many different ways can we group the children?

In this case, $n = 10, r_1 = 3, r_2 = 4, r_3 = 3$. Note that $n = r_1 + r_2 + r_3$. Filling in the formula, we get $\frac{10!}{3! * 4! * 3!}$.



Partition Rule: A technique for counting the number of possible divisions of some set into various unequal groups.



2.7 Exercises

Solutions to these exercises can be found in Appendix A.2 on page 244.

1. *Problem:* Sam buys a box of mixed chocolates. The box contains 3 nut varieties, 5 truffle varieties, 3 caramels, and 2 hard chocolates. Suzie opens the box first and eats 4 chocolates; but she doesn't like truffles or hard chocolates, so she won't eat any of those. Sam doesn't like nut varieties or hard chocolates. How many chocolates are left that Sam might like?
2. *Problem:* Given that $|A| = 5$ and $|B| = 11$, but not knowing the details of the contents, what is $|A \cap B|$?
3. *Problem:* Given that $|A| = 4$ and $|B| = 6$, but not knowing the details of the contents, what is $|A \cup B|$?
4. *Problem:* An IPv4 network address consists of 32 bits. Each bit has two possibilities (0 or 1). What is the maximum theoretical number of possible IPv4 addresses?
5. *Problem:* There are 6 boxes numbered 1, 2, 3, 4, 5, and 6. Each box is to be filled up either with a red or a green ball in such a way that at least 1 box contains a green ball and the boxes containing green balls are consecutively numbered. How many different arrangements are possible?
6. *Problem:* How many different four letter words can be formed (the words need not be meaningful) using the letters of the word MEDITERRANEAN such that the first letter is E and the last letter is R?
7. *Problem:* How many ways can a class of six people be split into two equal groups if there is no distinction between the groups?
8. *Problem:* A restaurant offers the following menu: select either an entree or a burger. There are three entrees available, and each entree comes with two sides. There are four sides to choose from. We can choose two of the same side, if desired. If we choose the burger, on the other hand, there are five different burgers available, but burgers do not come with any sides. How many different selections are possible?

Chapter 3

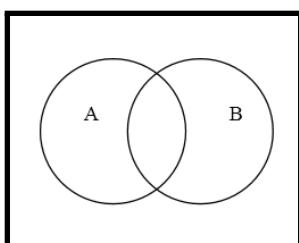
Venn Diagrams

Venn diagrams are a visual tool for describing how sets are related. A Venn diagram consists of a rectangular area which encompasses the universe, \mathcal{U} . Each set under consideration appears, usually as a circle or oval, within this rectangle. The various sets are arranged in such a way that all possible set relationships could be diagrammed. Thus, the various sets must overlap in all possible ways.

Venn diagrams are particularly useful for comparing two set expressions to see if they are equal (equivalent set expressions will have the same Venn diagram), as well as for simplifying set expressions, a concept discussed further in the next chapter.

Venn diagrams do not depend on particular details about the universe or the sets being visualized: they are constructed entirely from a set expression. For now, let's imagine that we have two sets A and B within the universe \mathcal{U} .

The empty set \emptyset is visualized by filling in nothing:



Venn Diagram: a visual representation of a set expression.



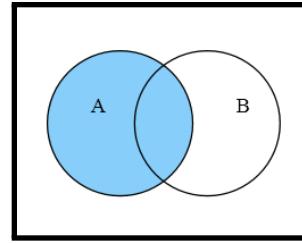
$\sum_{n=1}^{\infty} \frac{1}{n^2}$ If a Venn diagram visualizes n sets, then it must have 2^n distinct regions.

Venn diagrams in this book are produced using the Venn Visualizer software found at <http://www.sf.net/projects/vennvis>

Notice that the sets A and B each have an overlapping circle, but these areas are not filled in because we have neither the set A nor the set B . The area outside of the circles represents those parts of the universe in neither set A nor set B . Again, because our expression is \emptyset , nothing is filled in.

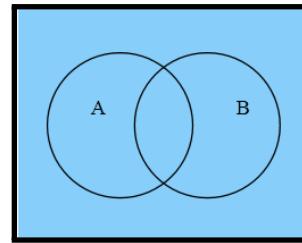
If we visualize the set A , we fill in just the A circle:

Although part of B is filled in (due to the overlap with A), this does not necessarily guarantee that a specific set B does contain elements in common with A ; it only says that if the set B does contain elements in common with A , then by selecting the set A we get those elements as well.



Notice that the entire A circle is filled in, including those portions which overlap with B (the middle section) and those portions which only appear in A .

If we visualize the universe \mathcal{U} , then the entire diagram, including all sets, is filled in:

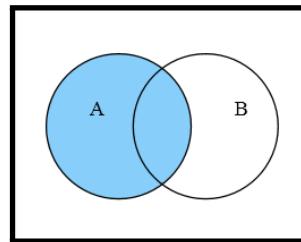


3.1 Set Operations

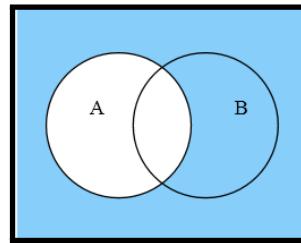
Each set operation can be defined as a function with takes one (complement) or two (union and intersection) Venn diagrams and produces a new diagram. The complement operator “inverts” the fill on a Venn diagram; whatever was filled in becomes blank, and whatever was blank becomes filled in. The union operator takes whatever was filled in either or both of the original diagrams and fills that in the new diagram. The

intersection operator takes whatever was filled in both of the original diagrams and fills that in the new diagram.

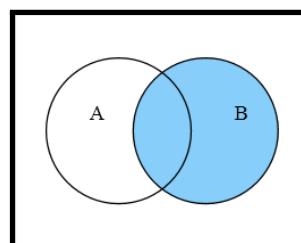
Given that the set A is visualized by:



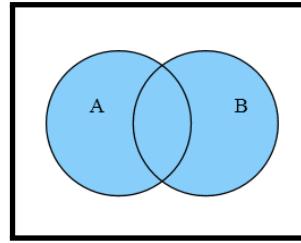
Then the set A' (complement of A) is visualized by:



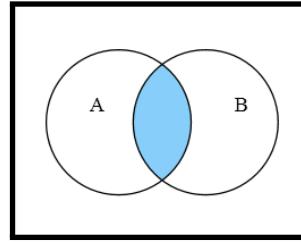
If we consider the visualization of A (shown above) and the visualization of B :



Then the operation of $A \cup B$ (union of A with B), defined as all elements in A together with all elements in B (including elements in both) is visualized by:



The operation of $A \cap B$ (intersection of A with B), defined as all elements that appear in both A and B , is visualized by:



3.2 Other Set Operations

Besides the standard operations of complement, union, and intersection, several other convenient set operations may be defined.



Set Difference: the set of all elements that appear in the first set but not the second set.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

$$A \setminus B = B' \cap A$$

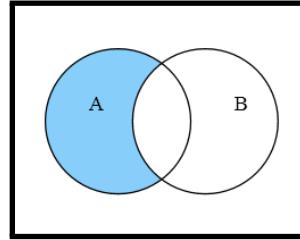


Symmetric Difference: the set of all elements that appear in the first set or the second set, but not both.

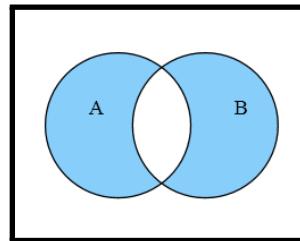
$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

$$A \Delta B = (A \cup B) \cap (A \cap B)'$$

The set difference of two sets, written $A \setminus B$, is all elements in A that do not appear in B . The set difference may also be called the relative complement. The set difference $A \setminus B$ can be visualized by:



The symmetric difference of two sets, written $A \Delta B$, is all the elements in either A or B , but not both. In logic, this operation is commonly referred to as exclusive or. The symmetric difference $A \Delta B$ can be visualized by:



3.3 Creating a Venn Diagram

Your best choice for creating a Venn diagram is to use a computer program to assist you. However, if no program is available, it is certainly possible to create a

Venn diagram for any set expression by hand. The general procedure for creating a Venn diagram by hand is:

1. Following order of operations, start at the innermost operation.
2. If the innermost item is a set symbol, fill the appropriate circle for that symbol.
3. When dealing with a union, overlap the left and right Venn diagrams. Create a new Venn diagram filling in all areas filled in on either diagram.
4. When dealing with an intersection, overlap the left and right Venn diagrams. Create a new Venn diagram filling in only the areas filled in on both diagrams.
5. When dealing with complement, create a new Venn diagram filling in all areas not filled in on the original diagram.
6. Repeat this process, moving upwards and outwards, until all operations have been processed.



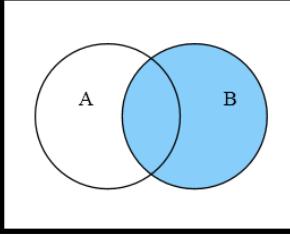
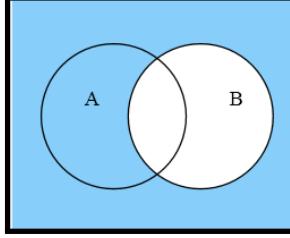
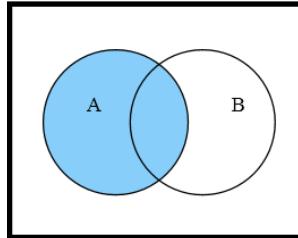
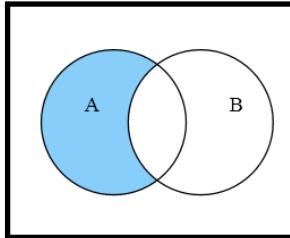
In many cases, you may be able to sketch a Venn diagram by simply reasoning through these steps in your head without excessive scratch paper or intermediate results.



Turning a set expression into words can also help with visualization. For example, $A \cap B'$ can be read as “everything in A that’s also not found in B ”.

■ **Example 3.1** • For example, let’s create by hand the Venn diagram for the set expression $A \cap B'$.



<p>First, find the inner-most set B:</p> 	<p>The complement is the next step, so fill in the opposite of everything that was filled in before:</p> 
<p>Next, determine the set A:</p> 	<p>Finally, overlap A with B' and, because it is intersection, fill in those areas in common:</p> 

3.4 Higher Order Venn Diagrams

As the number of sets to be visualized increases, the diagram complexity grows exponentially. In order to provide for all possibilities of n sets, we must consider that each set is either included or excluded. Because each yes or no refers to a different set from the others, order is important, and repetition (multiple yes or multiple no) is allowed. That is how to discover that to represent n sets, you must have 2^n regions. With just two sets, as shown above, only four regions are needed: the universe outside of both sets; just the A part, just the B part, and the intersection sliver of both A and B .

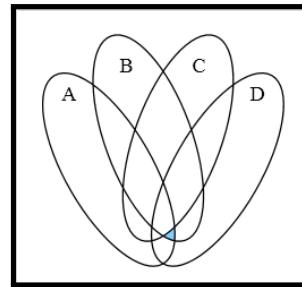
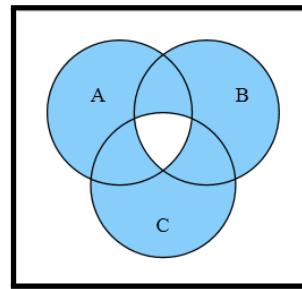
A Venn diagram of three sets is commonly represented with three overlapping circles. Note that there are eight regions; including none of the sets, each set by itself,



every pair of sets, and the intersection of all three sets. This diagram visualizes the expression $(A \cup B \cup C) \cap (A \cap B \cap C)'$.

Three sets is the highest number normally done by hand; diagrams of more than three sets become more difficult to reason about and generally less helpful. However, visualizing four, five, and even more sets is theoretically possible. Plain overlapping circles no longer suffice for diagrams of these orders, and other shapes are used: ovals, rectangles, or a mixture of various shapes.

Here is a Venn diagram of four sets. Notice the tiny area near the bottom which is filled in. This diagram is visualizing $A \cap B \cap C' \cap D$. The relative size of the areas is not significant in any particular way: it does not imply that the number of elements in this area is smaller than the number of elements in any other area. It is just an artifact of the particular visualization technique.



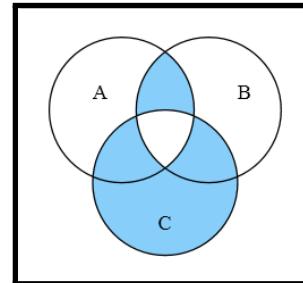
3.5 Exercises

Solutions to these exercises can be found in Appendix A.3 on page 248.

1. *Problem:* Draw a Venn diagram to visualize $A' \cup B' \cap C$.
2. *Problem:* Determine if $A' \cap B' = (A \cup B)'$.

Let the universe be customers of a bank. Let the set $A = \{x : x \text{ has a savings account}\}$, $B = \{x : x \text{ is a preferred customer}\}$, and $C = \{x : x \text{ has a checking account}\}$.

3. *Problem:* Draw a Venn diagram illustrating preferred customers who don't have a savings account.
4. *Problem:* Draw a Venn diagram illustrating non-preferred customers, except those who hold both a checking and savings account.
5. *Problem:* Determine a set expression that matches the following Venn diagram:



6. *Problem:* Symmetric difference is defined earlier in this chapter as $A \Delta B = (A \cup B) \cap (A \cap B)'$. An alternative definition of symmetric difference is given by $A \Delta B = (A \setminus B) \cup (B \setminus A)$. Prove that these two definitions are equivalent.
7. *Problem:* How many different Venn diagrams of two sets are possible?
8. *Problem:* There are four possible Venn diagrams of one set. Enumerate set expressions for each.

Simplifying Set Expressions

Set expressions derived from other sources, such as Venn diagrams, can be needlessly complex. Complex set expressions may result in difficult to understand and maintain computer programs. If these expressions can be simplified before they are used as a basis for program logic, then the implemented program logic will be shorter and easier to understand.

In order to assist in simplification, we'll take advantage of a variety of algebraic properties of set expressions that allow us to transform expressions meeting a certain pattern into another expression, guaranteed to be equivalent.

4.1 Algebra Laws of Sets

In these laws, the variables x , y , and z may represent any set expression or single set variable. Multiple uses of the same variable in one rule must all be identical expressions. For example, if we take the simple law $x \cup x = x$, this could be applied to the set expression $(A \cup (B \cap C)) \cup (A \cup (B \cap C))$. In that case, we would let $x = (A \cup (B \cap C))$, find that the set expression met the pattern, and reduce it to simply $A \cup (B \cap C)$. In all



Many of the set algebra rules mirror arithmetic algebra rules, and go by the same or similar names.

cases, a rule can be applied in either direction; either transforming an expression that meets the left pattern into the right pattern, or vice versa.

Idempotent Laws

$$1a. \quad x \cup x = x$$

$$1b. \quad x \cap x = x$$

Associative Laws

$$2a. \quad (x \cup y) \cup z = x \cup (y \cup z)$$

$$2b. \quad (x \cap y) \cap z = x \cap (y \cap z)$$

Commutative Laws

$$3a. \quad x \cup y = y \cup x$$

$$3b. \quad x \cap y = y \cap x$$

Distributive Laws

$$4a. \quad x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$$

$$4b. \quad x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$$

Identity Laws

$$5a. \quad x \cup \emptyset = x$$

$$5b. \quad x \cap U = x$$

$$5c. \quad x \cup U = U$$

$$5d. \quad x \cap \emptyset = \emptyset$$

Double Negation Law

$$6. \quad x'' = x$$

Complement Laws

$$7a. \quad x \cup x' = U$$

$$7b. \quad x \cap x' = \emptyset$$

$$7c. \quad U' = \emptyset$$

$$7d. \quad \emptyset' = U$$

DeMorgan's Laws

$$8a. \quad (x \cup y)' = x' \cap y'$$

$$8b. \quad (x \cap y)' = x' \cup y'$$

Absorption Laws

$$9a. \quad x \cup (x \cap y) = x$$

$$9b. \quad x \cap (x \cup y) = x$$

Any of these laws can be applied to a set expression or complete subexpression, and the transformation is guaranteed to be equivalent.

4.2 Explanation of Laws

 **Idempotent:** a function that, given two equal values, returns that value as the result.

 **Associative:** a function that, in a sequence of that function, the arrangement of parentheses can be changed without changing the final result.

The laws themselves are sufficient for simplifying set expressions, however, it may be useful to understand the meaning and significance behind each law.

The idempotent laws state that the union or intersection of a set with itself is simply that set. Thus, any such union or intersection appearing in a set expression can be cut in half to just show the main expression.



The associative law states a series of unions, or a series of intersections, can be done in any order (remember parentheses are done first, so they control the order of operations). It is important to note that the associative law only holds when all the operations in question are unions or all intersections; a mix of unions and intersections does not generally hold under association; $A \cup (B \cap C) \neq (A \cup B) \cap C$.

The commutative law, in this case logically extending the associative law, states that union and intersection consider their left and right parameters equally; there is no significance to being on the left or on the right. This allows us to reorder set parameters in a series of all unions or all intersections. We cannot reorder across different operations though; $A \cup B \cap C \neq C \cup B \cap A$.

The distributive law holds when an operation can be distributed into parentheses. This law is mostly used in the backwards direction to reduce an expression. Different operators are being used within and outside the parentheses, so they can't simply be shifted around as in the associative or commutative laws. Instead, the correct operation must still be applied, but it can be applied to each element within the parentheses and then the final results combined, or vice versa.

The identity law describes situations in which a set expression is applied to an operation that "does nothing", in some sense. There are two main types of identity functions shown here: the type that preserve a set expression, and the type that consume one. The preservation laws show that we can union a set with the empty set, or intersect a set with the universe, and retain the same original set. The consumption laws show that unioning a set with the universe, or intersecting it with the empty set, will destroy the original set and leave just the universe, or the empty set, respectively.

The double negation, or involution, law indicates that if we take the complement of any set, and then the complement of that, we get back where we started. No matter how many complement operators are present, they can always be removed in pairs: $A''''' = (((A'')'')' = A'$, for example.



Make sure you exactly match the unions and intersections in a rule to your set expression before you apply it. Some rules only work on certain arrangements of unions and/or intersections.



Commutative: a function whose order of parameters can be swapped without changing the final result.



Distributive: a function that, applied to a parentheses group can be distributed into and applied individually to each element within that group.



Identity: a function that, given any parameter, returns that value as the result.



Double Negation: the opposite of the opposite of any expression is itself.



For any algebraic law on sets, that law has a dual law which is found by replacing all \cup with \cap , all \cap with \cup , all \mathcal{U} with \emptyset , and all \emptyset with \mathcal{U} . For example, given the law $x \cup \mathcal{U} = \mathcal{U}$, it must also be true that $x \cap \emptyset = \emptyset$.





DeMorgan's Law: a complement can be distributed into an expression if the union and intersections are flipped.

The complement laws describe how the complement, or opposite, of certain sets behave. For example, the universe and empty set are complements of each other. Likewise, any set and its complement can be combined either with union to form the universe, or with intersection to form the empty set.

DeMorgan's Law is an interesting rule that originates in Boolean logic (we'll see more of this later). It says that if an element is found in neither A nor B , represented by $(A \cup B)'$, then that element is not found in A , and it is not found in B , represented by $A' \cap B'$.

The absorption law can be described with subset reasoning: If $A \subseteq B$, then $A \cup B$ must simply be B , because all elements in A are already in B , so the union contributes nothing new. How can we be sure, in a general set expression, that $A \subseteq B$? If we took $B \cap A$, then that result must be a subset of B because it could only contain elements that appear in B . So in the expression $B \cup (B \cap A)$, the latter part $B \cap A$ must be a subset of the first part B . Since we are combining them with union, the latter part must contribute nothing new. So that whole expression can be represented as just B .

4.3 Applying Algebra Laws

The various set algebra laws can be used to transform any given set expression into many other equivalent set expressions. In general, a common goal is to reduce the number of operations to the smallest count possible. Other less common goals might be to remove certain operators completely; it turns out that all set expressions can be represented with complement and either union or intersection, but not both.

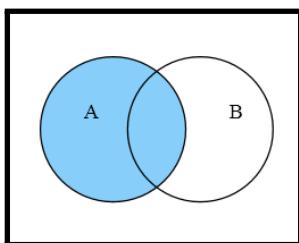


If you know you need to get rid of certain symbols from the expression, focus on that part of the expression when looking for appropriate laws.

Assuming the goal is to simplify the expression as much as possible, a good first step might be to create a Venn diagram. The Venn diagram may make the final reduction expression obvious, and it can also serve as a check to ensure that equivalence is maintained throughout the process. At all times, the Venn diagram should be the same for each simplification step.



■ **Example 4.1** • Let's consider first a simple example of $(B \cap B') \cup A$. If we visualize this as a Venn diagram, we come up with:



Therefore, we know right away that this expression must simplify to A . But how can we prove it using the laws? Unfortunately, there is no automatic technique to select which law to use in simplification: it is a process of trial and error. Different sequences of applications may lead in longer or shorter paths, but ultimately you should ensure that an unbroken chain is followed until the simplest form is reached.

In this case, we know that we need to get rid of the B symbols entirely. Consulting the laws, therefore, we find a matching pattern in complement law 7b, which tells us that $B \cap B' = \emptyset$. We can then substitute this part into the original expression to get the new expression $\emptyset \cup A$. This expression is guaranteed to be equivalent, and have the same Venn diagram, as the original expression. Try it and make sure.

Whenever the empty set or the universe appears in an expression, we should look to the identity laws to see if it can be removed. In this case, if we first flip the subparts using commutative law 3a, our expression matches identity law 5a. This eliminates the empty set and leaves us with our final simplified expression A .

A simplification can be documented as a step-by-step procedure, showing at each step the expression so far and the law being applied to reach the next step. When subexpressions are being transformed, it may be helpful to underline them to help show the reader exactly what is happening.

It's easy to skip specifically applying associative and commutative laws and "automatically" rearrange set expressions as needed. However, there is danger here as not all rearrangements are valid. A mistaken invalid rearrangement can cause the entire simplification to break down.

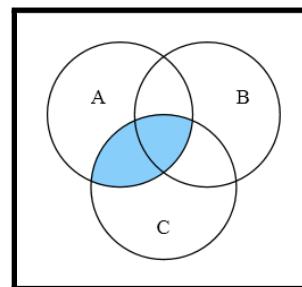


Remember the goal of the step-by-step procedure is to serve as a proof that the transformation is valid. Therefore, include as much detail as is necessary to convince the reader that the laws have been followed correctly.

1. $(B \cap B') \cup A$ Initial Set
2. $\underline{(B \cap B')} \cup A$ Complement Law 7b
3. $\emptyset \cup A$ Commutative Law 3a
4. $A \cup \emptyset$ Identity Law 5a
5. A Final Expression

■

■ **Example 4.2** • Let's now consider a more complicated case. How about the expression $(B \cup C) \cap (C \cap A)$. If we visualize this expression as a Venn diagram, we find it is simply the intersection $C \cap A$:



To begin solving this problem, we actually have to make it a little worse first.

1. $(B \cup C) \cap (C \cap A)$ Initial Set
2. $(C \cap A) \cap (B \cup C)$ Commutative Law 3b
3. $((C \cap A) \cap B) \cup ((C \cap A) \cap C)$ Distributive Law 4b
 $(C \cap A)$ is the x in the rule)

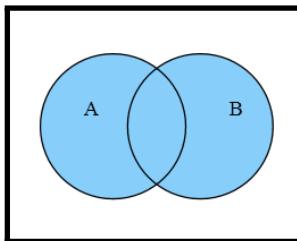
Notice the application of the distributive law here makes the expression more complex; however, this complexity is necessary to open doors to simplification. In particular, by expanding the expression first, we can take advantage of the powerful absorption law to eliminate the unused portion.

4. $((C \cap A) \cap B) \cup ((C \cap A) \cap C)$ Current Set
5. $((C \cap A) \cap B) \cup \underline{((C \cap A) \cap C)}$ Commutative Law 3b
6. $((C \cap A) \cap B) \cup \underline{(C \cap (C \cap A))}$ Associative Law 2b
7. $((C \cap A) \cap B) \cup \underline{((C \cap C) \cap A)}$ Idempotent Law 1b
8. $((C \cap A) \cap B) \cup \underline{(C \cap A)}$ Commutative Law 3a
9. $(C \cap A) \cup ((C \cap A) \cap B)$ Absorption Law 9a
10. $C \cap A$ Final Expression

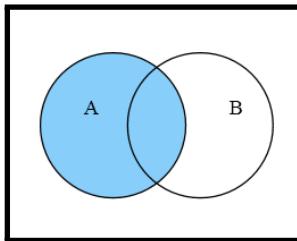
■ **Example 4.3** • What if you have the expression $B \cup B' \cap A$? The immediately obvious law to apply is the



complement law 7a on the first portion. This, however, creates a problem. First, let's visualize $B \cup B' \cap A$:



Apply the complement law 7a to the first portion of the expression, yielding $\mathcal{U} \cap A$, which has the Venn diagram:



The set laws are guaranteed to never change the meaning of a set expression; that is, whenever a set law is applied the new expression will be equivalent to the old expression. In this case, however, these two expressions are clearly not equivalent. The problem here is failure to obey order of operations. Union and intersection do not have the same precedence, which means you must imagine parentheses grouping around the higher precedence operation: $B \cup (B' \cap A)$. In order to perform the complement as specified, the parentheses would have to be moved to $(B \cup B') \cap A$, however, this move is not supported by any associative law. Therefore, the simplification is invalid. ■

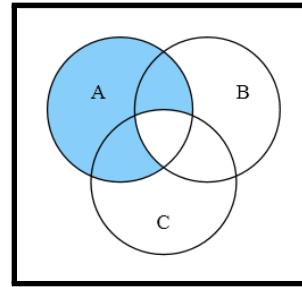
Be careful of order of operations; when unions and intersections are mixed, remember that intersection occurs first. Write in parentheses to enforce correct ordering.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

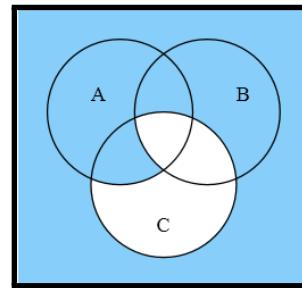
$x \cup y \cap z = x \cup (y \cap z).$
However, $x \cup y \cap z \neq (x \cup y) \cap z.$

- **Example 4.4** • To consider another example where order of operations could trip up a transformation, consider the expression $A \cap (B \cap C)'$. This expression can be diagrammed as:





If we apply DeMorgan's Law to the latter half of the expression, it seems to directly map $A \cap B' \cup C'$. However, this is not correct. Using order of operations, we would evaluate that expression as $(A \cap B')$ $\cup C'$ which gives the diagram:



Even though parentheses were not specified in the rule, each rule applies atomically, and the result must follow the same order of operations as the original subexpression. To ensure a subexpression is atomic, we must be able to put parentheses around it without changing the meaning of the whole expression. Likewise, the rule result should have parentheses until we determine they are not needed. Thus, the correct application of DeMorgan's Law is $A \cap (B \cap C)' = A \cap (B' \cup C')$. ■

4.4 Exercises

Solutions to these exercises can be found in Appendix A.4 on page 256.

1. *Problem:* Simplify the set expression $(A' \cup B')'$.
2. *Problem:* An incorrect attempt at simplifying $B \cup B' \cap A$ was shown earlier in the chapter. Show a correct simplification for this expression.
3. *Problem:* Simplify the set expression $A \cap B \cap A' \cup A \cup B' \cup A'$.
4. *Problem:* In the previous chapter, a certain Venn diagram was found to have the set expression $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (B \cap C \cap A') \cup (C \cap (A \cap B))'$. Simplify this expression.
5. *Problem:* Prove that $(A \cap B') \cup (A \cap B) = A \cup (B \cap B')$.
6. *Problem:* Prove the absorption law (9a)
 $x \cup (x \cap y) = x$.
7. *Problem:* Prove the absorption law (9b)
 $x \cap (x \cup y) = x$.

8. *Problem:* Megan is writing a video game which needs to process space ships with various weapons. Let the universe be all space ships in play, the set L be all space ships with lasers, the set M be all space ships with missiles, and the set D be all space ships with death rays. Any space ships may be configured with any or all weapons (or no weapons).

Megan needs to write a program to find all space ships which meet one or more of the following criteria:

- (a) The ship has both missiles and lasers.
- (b) The ship has death rays or lasers, but not both.
- (c) The ship has missiles.
- (d) The ship has lasers, missiles, and death rays.

Devise a single simplified set expression to find the ships Megan is looking for.

PART II

Boolean Logic

5 Logical Operators and Truth Tables	37	7 Decision Tables	62
5.1 Truth Tables	37	7.1 Boolean Decision Tables	62
5.2 Logical Operators	38	7.2 Indifferent Conditions	65
5.3 Logical Expressions	39	7.3 Relationship to Boolean Expressions	66
5.4 More Logical Operators	41	7.4 Multi-valued Conditions	67
5.5 Relationship to Set Expressions	43	7.5 Impossible Conditions	69
5.6 Reverse-Engineering a Truth Table	44	7.6 Exercises	71
5.7 Operator Summary	46	8 Logic Circuits	73
5.8 Exercises	47	8.1 Fundamental Logic Gates	73
6 Manipulating Logical Expressions	48	8.2 Logic Circuits	75
6.1 Satisfiability	48	8.3 Compound and Universal Gates	76
6.2 Boolean Algebra Laws	49	8.4 Converting to Boolean Expressions	78
6.3 Use of the Equivalence Operator	51	8.5 Converting from Boolean Expressions	80
6.4 Applying Algebra Laws	54	8.6 Partial Circuit Reuse	83
6.5 Use of the Assignment Operator	55	8.7 Disjunctive Normal Form	85
6.6 Relationship to Natural Language	57	8.8 Exercises	87
6.7 Exercises	61		

Chapter 5

Logical Operators and Truth Tables

Sets are important for considering collections of objects to be processed. Computers, however, usually process elements one at a time. Internally, consideration of any object breaks down into a series of yes/no (or true/false) questions. These questions form the fundamental groundwork of computation. Boolean expressions (named after George Boole, who discovered these techniques), are closely related to set expressions, but consider only singular true/false values rather than sets of objects.

5.1 Truth Tables

Set operations may be described, perhaps in some vagueness, by Venn diagrams. Boolean operations, dealing with only the values true and false (often abbreviated as T and F), can be described exactly by using truth tables. All Boolean operations are formally defined either by truth tables or by reference to other Boolean operations. Truth tables are composed of a series of columns, one for each input variable, and one final output column. Each possible permutation of input values constitutes a row.

Two Boolean expressions can be checked for equiva-



Boolean: a 2-valued object, whose values are usually represented as yes/no, true/false, or 1/0.



Truth Table: a table indicating the true/false result of a Boolean expression for all possible permutations of input values.



If a Boolean expression has n variables, then its truth table will be $n + 1$ columns and 2^n rows.

 Simple and complex truth tables for any Boolean expression can be created with a variety of online tools, including a nice Java implementation found at <http://www.brian-borowski.com/Software/Truth/>

 An easy online truth table generator, although not as pretty as the above, can be found at <http://turner.faculty.swau.edu/mathematics/materialslibrary/truth/>

 To be correct, a truth table must list the result of all possible permutations of input values; and it must not contain a contradictory claim. The best way to ensure these conditions are met is to verify that a truth table has the correct number of rows (2^n for n input values) and that each row's input values are distinct.

 The ordering of the rows and columns are not significant, and different tools and authors may order the rows and/or columns in different ways. You must check the values specified in each row, and match the variable specified for each column, to ensure you have the right entry.

lence by creating a truth table for each expression. Two equivalent Boolean expressions will always have the same truth table.

5.2 Logical Operators

Boolean expressions are built on three fundamental operators: AND, OR, and NOT.

The operator AND is represented with the caret \wedge , or the dot \cdot or $*$, or $\&$ between two Boolean values or subexpressions. The AND operator returns true when both of the inputs are true, and false in any other case. Here is the truth table for $a \wedge b$:

a	b	$a \wedge b$
T	T	T
T	F	F
F	T	F
F	F	F

If we had a pair of Boolean values representing a and b , we could reference the truth table to find out what the result of $a \wedge b$ is. Arbitrarily large Boolean expressions can be solved step by step in this way.

The operator OR is represented with the inverted caret \vee , or $+$, or $|$, between two Boolean values or subexpressions. The OR operator returns true when either or both of the inputs are true. It returns false only when both of the inputs are false. Here is the truth table for $a \vee b$:

a	b	$a \vee b$
T	T	T
T	F	T
F	T	T
F	F	F

The operator NOT is represented with the symbol \neg , or with the tilde \sim , or $!$, in front of a Boolean value. The



NOT operator returns true when the input is false, and it returns false when the input is true. In other words, the NOT operator inverts the input. Here is the truth table for $\neg a$:

a	$\neg a$
T	F
F	T

Operations such as AND, OR, and NOT are often described as being either binary or unary; binary operations (AND and OR) take two inputs, while unary operations (NOT) take only one input.

The NOT operator takes only one input value, so its truth table consists only of two rows. The AND and OR operators both took two input values, requiring larger truth tables to fully describe.

 *Binary Operator:* an operator that takes two inputs.

 *Unary Operator:* an operator that takes only one input.

5.3 Logical Expressions

A fairly ordinary order of operations applies with Boolean expressions. NOT has the highest precedence, followed by AND, with OR having the lowest precedence of the three. Therefore, the expression $\neg a \wedge b$ is the same as $(\neg a) \wedge b$, but not the same as $\neg(a \wedge b)$. Likewise, $a \vee b \wedge c$ is the same as $a \vee (b \wedge c)$, but not the same as $(a \vee b) \wedge c$.

To create the truth table for any logical expression, first count the number of input variables in the expression. Create a table with the appropriate number of rows and columns for those input variables. Fill in the input part of the rows. Then, for each row, solve the expression from the inside out using order of operations. Fill in the final result in the output column, and proceed to the next row.

■ **Example 5.1** • For example, let's construct the truth table for $\neg a \wedge b$. We first note that there are two distinct Boolean variables a and b in the expression. So we will create a table with 3 columns (a , b , and the output) and $2^2 = 4$ rows.



a	b	$\neg a \wedge b$
T	T	
T	F	
F	T	
F	F	

Each row will tell us what the specific input values are for those variables. Then, we'll plug them into the expression and solve, keeping order of operations in mind. First, both a and b will be true. The innermost operation is $\neg a$, which will be false (refer to the truth table for \neg if in doubt). Now we have $\neg a$ is false and b is true. These two are combined with an AND, which will return false, which is the final result for this row.

If the solution to a given expression with certain values is difficult to manage mentally, a chart can be used. First, write out the expression and place the known variable values underneath each instance of a variable. Consider the case for the second row, where a is true and b is false. Build the result of the expression from the inside out, crossing out values as they are consumed.

$$\begin{array}{ccccc} \neg & a & \wedge & b \\ T & & & F \\ F & \textcolor{orange}{T} & & F \\ \textcolor{orange}{F} & \textcolor{orange}{T} & F & \textcolor{orange}{F} \end{array}$$

Thus, the outcome given the inputs $a = T$ and $b = F$ is false. This process continues to determine the result of each row. The final truth table is:

a	b	$\neg a \wedge b$
T	T	F
T	F	F
F	T	T
F	F	F

 *Short-Circuit Evaluation:* portions of a Boolean expression may be skipped (unevaluated) if it is known that their value will not affect the final result.

In some expressions a technique known as short-circuiting may be helpful. For example, consider the



expression $a \wedge (b \vee \neg(c \wedge d) \vee (b \wedge d))$. In this expression, if we knew that a was false, we could skip the entire right-hand side and go straight to the result: false. Short-circuiting is specifically possible with two operators: AND, if one side is false the entire expression is false; and OR, if one side is true the entire expression is true.

5.4 More Logical Operators

Based on the three fundamental operators, several more common operators are defined. These operators may be defined either with a truth table, or as an expression built of the fundamental operators.

The OR operator, as previously defined, allows either or both of the inputs to be true. A modified version, the Exclusive-OR, or XOR, operator returns true if exactly one of its inputs are true. This can be read as one or the other, but not both. The XOR operator is represented with the circled plus symbol \oplus , and can be defined as $a \oplus b = (a \vee b) \wedge \neg(a \wedge b)$. Here is the truth table for $a \oplus b$:

a	b	$a \oplus b$
T	T	F
T	F	T
F	T	T
F	F	F

The implication operator is true if, given a condition, the result is also guaranteed. This operator takes some consideration to fully understand. We'll first give the definition and truth table, and then consider carefully every row to understand why the result makes sense. The implication, given with the symbol \rightarrow , can be defined as $a \rightarrow b = \neg a \vee b$. Here is the truth table for $a \rightarrow b$:

 *Implication:* an expression “ a implies b ” is true if, whenever a is true, b is guaranteed to be true. If a is false, b is irrelevant and the implication is automatically true.

 Implication is the only Boolean operator shown here which is not symmetric; that is, if $a \rightarrow b$ is true, this does *not* mean that $b \rightarrow a$ is also true.

a	b	$a \rightarrow b$
T	T	T
T	F	F
F	T	T
F	F	T

This truth table seems at odds with the most intuitive definition of implication. We'll look at each row one at a time. In the first row, if both a and b are true, it seems reasonable that the claim " a implies b " is upheld, so the result is true. In the second row, a is true but b is false. This contradicts the claim that " a implies b ", as we have a true value for a but a false value for b . So the implication itself is false in this case. The third and fourth cases can be taken together: in both cases, a is false. This means the statement " a implies b " is vacuously true: because we don't have a , we can't make any claim on the value of b .

For example, if we said "Rain implies wet pavement", but there was no rain, then the presence or absence of wet pavement would not invalidate the statement. However, if there was rain, then the pavement would have to be wet, otherwise the claim "rain implies wet pavement" becomes itself false.

A final operator to consider is the logical biconditional, perhaps easiest understood as equality. The logical biconditional operator, written with the double sided arrow \leftrightarrow , or sometimes the equivalence symbol \equiv , is true if both of its inputs are equal (regardless of whether those inputs are true or false). This operator can be defined with $a \leftrightarrow b = \neg(a \oplus b)$. The reason for the name biconditional, and the choice of double arrow comes from the alternative definition $a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$.

a	b	$a \leftrightarrow b$
T	T	T
T	F	F
F	T	F
F	F	T



5.5 Relationship to Set Expressions

Boolean expressions and set expressions share a close relationship. A set expression which describes a set through the use of unions and intersections can be translated directly into a Boolean expression which describes the same relationship for a single element that may be a member of the set(s).

Set Notation	Meaning	Boolean Notation	Meaning
A, B, C, \dots	An unordered collection of elements	a, b, c, \dots	True/false value indicating if some object is a member of the respective set. If a is true, then the object is a member of A .
$A \cup B$	All the elements in A together with all the elements in B	$a \vee b$	True if the object is in either the set A or B , or both. In other words, true if the object is in $A \cup B$.
$A \cap B$	All the elements that are in both A and B .	$a \wedge b$	True if the object is in both sets A and B . In other words, true if the object is in $A \cap B$.
A'	All the elements in the universe not found in A	$\neg a$	True if the object is not in the set A .
$A \Delta B$	All the elements in either A or B , but not both.	$a \oplus b$	True if the object is in the set A , or the set B , but not both. In other words, true if the object is in $A \Delta B$.

In general, set expressions may be translated directly into Boolean expressions by replacing \cup with \vee and \cap with \wedge . Complement ' $'$ must be replaced with \neg , but in set notation the complement follows the subexpression, whereas in Boolean notation the complement precedes the subexpression. Any \emptyset in the set expression becomes the constant False, and any \mathcal{U} in the set ex-

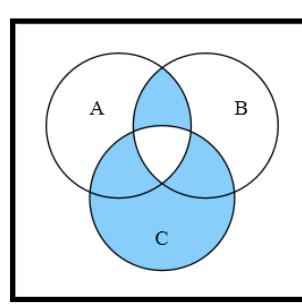


pression becomes the constant True. The set expression A' would be translated as $\neg a$; the set expression $(A \cup B)'$ would be translated as $\neg(a \vee b)$.

This truth table has eight rows because it has three variables, and $2^3 = 8$. It would not be possible to consider all regions/possibilities in less than eight rows.



Venn diagrams and truth tables are also directly correlated; the input values in the truth table describe each distinct region in a Venn diagram. If the result is true, then that region of the diagram is filled in. Consider the following Venn diagram and equivalent truth table. Notice that just C , $C \cap A \cap B'$, $C \cap B \cap A'$, and $A \cap B \cap C'$ are filled in on the Venn diagram, but other regions such as $A \cap B \cap C$ are not. Compare this to the truth table, which shows true for the filled in regions.



<i>a</i>	<i>b</i>	<i>c</i>	result
T	T	T	F
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

5.6 Reverse-Engineering a Truth Table

A mechanical procedure exists for taking a truth table of an unknown expression and determining an equivalent Boolean expression. This expression is likely not in simplest form, and should be simplified as we'll show later.

The procedure to create a Boolean expression from a truth table is:

1. Cross out all rows where the outcome is false.
2. For each remaining row: create a subexpression AND'ing each variable together, and place a NOT in front of those variables that are false in that



row. For example, if a is true, b is false, and c is true, the subexpression is $(a \wedge \neg b \wedge c)$.

3. Connect all the subexpressions together with ORs.

This procedure creates an expression in disjunctive normal form.

■ **Example 5.2** • For example, consider the following truth table:

<i>a</i>	<i>b</i>	<i>c</i>	result
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

What Boolean expression could match this truth table?
First, cross out all rows where the outcome is false.

<i>a</i>	<i>b</i>	<i>c</i>	result
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

For simplicity, we'll rewrite the partial truth table consisting only of rows with true outcomes.

<i>a</i>	<i>b</i>	<i>c</i>	result
T	T	T	T
T	F	F	T
F	T	T	T

Disjunctive Normal Form: a series of subexpressions all connected by OR operators. Each subexpression must consist of a series of distinct variables, each one possibly prefaced with a NOT, that are connected with ANDs. Also called Sum of Products..



For each row, create a conjunction (variables connected with ANDs).

a	b	c	Conjunction
T	T	T	$a \wedge b \wedge c$
T	F	F	$a \wedge \neg b \wedge \neg c$
F	T	T	$\neg a \wedge b \wedge c$

Finally, connect all the conjunctions together with ORs. The final Boolean expression is $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge c)$. ■

 Be sure to wrap each subexpression in parentheses to get the desired order of operations.

Some software programs which operate on Boolean expressions require disjunctive normal form (or its twin, conjunctive normal form). Any Boolean expression can be converted into disjunctive normal form by first creating its truth table, and then converting the truth table to disjunctive normal form as shown above.

5.7 Operator Summary

For completeness, here is a summary of the truth tables for all binary (two-input) operators described.

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$	$a \rightarrow b$	$a \leftrightarrow b$
T	T	T	T	F	T	T
T	F	F	T	T	F	F
F	T	F	T	T	T	F
F	F	F	F	F	T	T

5.8 Exercises

Solutions to these exercises can be found in Appendix A.5 on page 261.

1. *Problem:* Let r be true if the roses are red, let d be true if the daffodils are in bloom, and let c be true if the cucumbers are ripe. Write Boolean expressions for the following claims:
 - (a) The roses are red and the daffodils are in bloom.
 - (b) Either the cucumbers are ripe, or the daffodils are in bloom, but not both.
 - (c) Either the roses are red, or the daffodils are in bloom, or both.
 - (d) The roses are not red, nor are the daffodils in bloom, but at least the cucumbers are ripe.
 - (e) If the cucumbers are ripe, then either the roses must be red or the daffodils must be in bloom, or both.

 2. *Problem:* Create a truth table for the Boolean expression $(a \wedge b) \vee \neg(a \vee c)$.

 3. *Problem:* Create a truth table for the Boolean expression $a \wedge (T \vee b) \wedge (c \wedge F)$.

 4. *Problem:* Is $\neg a \wedge \neg b$ equivalent to $\neg(a \vee b)$?
-
5. *Problem:* Prove that $a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$.

 6. *Problem:* Consider the following two truth tables. Are these tables equivalent?

a	b	result
T	T	T
T	F	F
F	T	T
F	F	T

b	a	result
F	T	F
F	F	T
T	F	T
T	T	T

 7. *Problem:* Convert the set expression $A' \cup B' \cap (A \cup B)'$ into a Boolean expression.

 8. *Problem:* Find a Boolean expression matching the following truth table:

a	b	c	result
T	T	T	F
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

Manipulating Logical Expressions

A significant portion of computer program implementation is based on deciding under what conditions certain actions should be taken. These conditions are written with Boolean expressions. It is helpful to simplify these expressions as much as possible; primarily to help other programmers (and ourselves) understand what the program is doing, but also in certain cases for performance reasons.

Simplification of Boolean expressions follows the same rules as set expressions. However, there are some distinctly Boolean concerns.

6.1 Satisfiability



Contradiction: a Boolean expression that is never true regardless of input values.

A classic problem in programming is to determine, given a Boolean expression, if there is some permutation of values for the various variables that results in a true outcome. There are three possible outcomes: an expression could be a contradiction, it could be satisfiable, or it could be a tautology.

A common contradiction is indicated with the expression $a \wedge \neg a$. At one time, a particular Boolean variable may only have one value: true or false. In order for this

expression itself to be true, a would have to be both true and false at the same time. You can prove that an expression is a contradiction by constructing a truth table. If all of the outcome rows are false, the expression is a contradiction.

a	$a \wedge \neg a$
T	F
F	F

A tautology is the opposite of a contradiction: if all of the outcome rows are true, the expression is a tautology. A common tautology is indicated with the expression $a \vee \neg a$.

a	$a \vee \neg a$
T	T
F	T

Tautologies and contradictions are worthless in a computer program: they take up space and processing time, but will always come out either true or false (respectively), and so they add no benefit. In some cases, changes over time to a Boolean expression will cause it to become a tautology or contradiction. This case should be noticed and the expression can be removed.

These three attributes of Boolean expressions can also be described using sets. Let the universe \mathcal{U} be all possible Boolean expressions. Let $S = \{x : x \text{ is a satisfiable Boolean expression}\}$, $T = \{x : x \text{ is a tautology}\}$, $C = \{x : x \text{ is a contradiction}\}$. In that case, $\mathcal{U} = S \cup T \cup C$ (that is, all expressions are either satisfiable or contradiction), $S \cap C = \emptyset$ (that is, satisfiable and contradictory expressions are disjoint), $T \subset S$ (that is, all tautologies are satisfiable, but not all satisfiable expressions are tautologies).

 **Satisfiable:** a Boolean expression that is true for at least one permutation of input values.

 **Tautology:** a satisfiable Boolean expression that is always true.

 In a computer program, a block that says “if (some tautology) do (some behavior)” can simply be replaced by “do (some behavior)” since the condition will always be true.

 In a computer program, a block that says “if (some contradiction) do (some behavior)” can be removed entirely, since the condition will never be true. This case should be approached with care; sometimes the condition is mistaken and should be fixed to be satisfiable.

6.2 Boolean Algebra Laws

These laws mirror the set transformation laws, and the explanations are largely the same. The table is repro-

duced here, altered to show Boolean notation instead of set notation, for convenience. The only significant changes are the addition of laws regarding implication and exclusive or.

Idempotent Laws

$$1a. \quad x \vee x = x$$

$$1b. \quad x \wedge x = x$$

Associative Laws

$$2a. \quad (x \vee y) \vee z = x \vee (y \vee z)$$

$$2b. \quad (x \wedge y) \wedge z = x \wedge (y \wedge z)$$

$$2c. \quad (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

$$2d. \quad (x \leftrightarrow y) \leftrightarrow z = x \leftrightarrow (y \leftrightarrow z)$$

Commutative Laws

$$3a. \quad x \vee y = y \vee x$$

$$3b. \quad x \wedge y = y \wedge x$$

$$3c. \quad x \leftrightarrow y = y \leftrightarrow x$$

$$3d. \quad x \oplus y = y \oplus x$$

Distributive Laws

$$4a. \quad x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

$$4b. \quad x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

Identity Laws

$$5a. \quad x \vee F = x$$

$$5b. \quad x \wedge T = x$$

$$5c. \quad x \vee T = T$$

$$5d. \quad x \wedge F = F$$

$$5e. \quad x \leftrightarrow F = \neg x$$

$$5f. \quad x \leftrightarrow T = x$$

$$5g. \quad x \oplus F = x$$

$$5h. \quad x \oplus T = \neg x$$

Double Negation Law

$$6. \quad \neg\neg x = x$$

Complement Laws

$$7a. \quad x \vee \neg x = T$$

$$7b. \quad x \wedge \neg x = F$$

$$7c. \quad \neg T = F$$

$$7d. \quad \neg F = T$$

$$7e. \quad x \oplus \neg x = T$$

$$7f. \quad x \oplus x = F$$

$$7g. \quad x \leftrightarrow x = T$$

$$7h. \quad x \leftrightarrow \neg x = F$$

DeMorgan's Laws

$$8a. \quad \neg(x \vee y) = \neg x \wedge \neg y$$

$$8b. \quad \neg(x \wedge y) = \neg x \vee \neg y$$

Absorption Laws

$$9a. \quad x \vee (x \wedge y) = x$$

$$9b. \quad x \wedge (x \vee y) = x$$

Implication Laws

$$10a. \quad x \rightarrow (y \rightarrow z) = (x \wedge y) \rightarrow z$$

$$10b. \quad x \rightarrow y = \neg y \rightarrow \neg x$$

Caution! Unlike most operators, implication is NOT commutative. That is, $x \rightarrow y$ is NOT the same as $y \rightarrow x$.



6.3 Use of the Equivalence Operator

It is common for programmers to assign more (conceptual) weight to the equivalence operator (represented in most languages as `==`) than other operators, and as a result, not take full advantage of its capabilities. Like the other binary Boolean operators, it takes two parameters and returns a true or false value.

Note that in these examples, the variables given are all Boolean variables, holding only the possible values True or False, unless otherwise specified.

■ **Example 6.1** • Consider the expression $a = b \wedge (c \leftrightarrow d)$. This means that for a to be true, b must be true and c must equal d . A common (but not ideal) implementation technique for an expression of this type would be:

```
a = false
if (c == d) then
    if (b == true) then
        a = true
    end if
end if
```

Such an implementation is much longer than it needs to be. There is no necessary reason to split this expression into two separate conditions; and the specific comparison of b to true is extraneous (see rule 5f above). This code could be simplified substantially by following the Boolean expression faithfully:

```
a = b && (c == d)
```

Like any other operator, multiple uses of the equivalence operator in a single expression are acceptable.

■ **Example 6.2** • Consider the following implementation (unrelated to the previous example):

$\sum_{n=1}^{\infty} \frac{1}{n^2}$

In many programming languages, `¬` is represented by `!`, `∧` is represented by `&&` and `∨` is represented by `||`.

$\sum_{n=1}^{\infty} \frac{1}{n^2}$

In many programming languages, `x != y` is shorthand for `!(x == y)`.

```

a = true
if (c == d) then
    a = false
end if
if (b != c) then
    a = false
end if

```

The outcome of operations such as these may be inferred reasonably, or represented with a truth table. In this case b , c , and d are used to determine the outcome (they are input), and a is the output. It is possible that there may be several outputs, or that a single variable may serve in two roles, as both input and output.

Here is a truth table framework for this implementation:

b	c	d	a
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

If nothing happens, a remains true. If either $c \leftrightarrow d$ or $\neg(b \leftrightarrow c)$ is true, or both, a is set to false. We can fill in the rows where c and d are the same, as well as the rows where b and c are different, with the outcome of false. The other rows will be true.

First, here are the rows where c and d are the same:



<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>
T	T	T	F
T	T	F	
T	F	T	
T	F	F	F
F	T	T	F
F	T	F	
F	F	T	
F	F	F	F

Next, mark as false the rows where *b* and *c* are different.

<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>
T	T	T	F
T	T	F	
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	
F	F	T	
F	F	F	F

All remaining rows are marked as true:

<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>
T	T	T	F
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	T
F	F	F	F

We can represent this entire operator as $a = \neg((c \leftrightarrow d) \vee \neg(b \leftrightarrow c))$. This block of code can be reduced into a single Boolean expression:

```
a = ! ( (c == d) || (b != c) )
```





The same is true for other operators which return true/false values (most notably, the comparison operators $<$ and $>$).

■ **Example 6.3** • It is common to see an implementation such as:

```
if (a < b) then
    c = true
else
    c = false
end if
```

This should be simplified to:

$c = a < b$



6.4 Applying Algebra Laws

The application of the algebra laws largely follows the set simplification techniques. However, the limitation of Boolean values to only true and false provides some additional reasoning opportunities.

Consider the expression $a \leftrightarrow \neg a$. Intuitively, a could never equal $\neg a$, so this expression must be a contradiction, as shown in the Complement Laws. Can we prove it? Being a Boolean value, a must be either true or false. Consider both cases. If a is true, then the expression is $T \leftrightarrow \neg T$. Applying the NOT operator gives us $T \leftrightarrow F$. By rule 5e, this simplifies to $\neg T$, which is false. On the other hand, if a is false, then the expression is $F \leftrightarrow \neg F$. Applying the NOT operator gives us $F \leftrightarrow T$. By rule 5f, this simplifies directly to F . So regardless of the value of a , the result is always false; the statement is a contradiction.



Law of Excluded Middle: for any Boolean expression or value x , exactly one of x and $\neg x$ must be true, and one must be false.



■ Example 6.4 • More traditional simplifications are also possible. Consider the expression $(a \leftrightarrow T) \vee (a \leftrightarrow F)$. By the law of excluded middle, it seems clear this statement must be a tautology. A step by step simplification can aid in proving this claim.

1. $(a \leftrightarrow T) \vee (a \leftrightarrow F)$ Initial Expression
2. $\underline{(a \leftrightarrow T) \vee (a \leftrightarrow F)}$ Identity Law 5f
3. $a \vee \underline{(a \leftrightarrow F)}$ Identity Law 5e
4. $a \vee \neg a$ Complement Law 7a
5. T Final Expression

Simplification is useful for reducing the often overly complex disjunctive normal form expressions generated mechanically from truth tables.

■ Example 6.5 • Consider an expression from the last chapter: $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge c)$.

1. $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge c)$ Initial Expression
2. $\underline{(a \wedge b \wedge c)} \vee (a \wedge \neg b \wedge \neg c) \vee \underline{(\neg a \wedge b \wedge c)}$ Commutative Law 2a
3. $(\neg a \wedge b \wedge c) \vee (a \wedge b \wedge c) \vee (a \wedge \neg b \wedge \neg c)$ Distributive Law 4b
4. $((b \wedge c) \wedge (\neg a \vee a)) \vee (a \wedge \neg b \wedge \neg c)$ Complement Law 7a
5. $((b \wedge c) \wedge T) \vee (a \wedge \neg b \wedge \neg c)$ Identity Law 5b
6. $(b \wedge c) \vee (a \wedge \neg b \wedge \neg c)$ Final Expression

Add parentheses around sub-blocks when applying algebra rules to ensure order of operations stays correct.

6.5 Use of the Assignment Operator

Many programming languages allow the programmer to alter the value of a variable with an assignment operator, often $=$. These values may even be altered based on their current value, which can in various circumstances lead to confusion. A truth table can always be created for any Boolean operation, including those with mutable values, by having a column for the input value of a variable in the left group, and a column for the output value of a variable in the right group.

Many programmers consider altering a variable's value after it has been assigned, except in certain clear cases, to be a bad practice.

■ Example 6.6 • Consider the following implementation:

```
a = c || d
b = a && d
```

In this case, the input variables are c and d ; their values are assumed to have been set prior to entering this block. The output variables are a and b . It is true that a is also used as input to the expression for b , however, it is not considered an input variable because its value prior to the block is irrelevant.

Here is a framework for the truth table:

c	d	a	b
T	T		
T	F		
F	T		
F	F		

What if b was modified later in the program, and you replaced all instances of b with d ? This could cause incorrect modification of d instead. This is one of the reasons altering a variable's value after initialization can cause trouble.



We can fill in the truth table by devising logical expressions for each output variable, and evaluating them in the usual way. It may help to simplify any expression, if possible, before evaluating it. A simplified expression may also suggest a change in implementation. The logical expression for a is simply $a = c \vee d$. What about b ? We don't want to reference a , because our truth table doesn't define a as an input variable. We can substitute in the definition of a , however, and come up with $b = (c \vee d) \wedge d$. That can be simplified by the absorption law 9b to just $b = d$.

We can now fill in the truth table for each row, evaluating each output variable independently.

c	d	a	b
T	T	T	T
T	F	T	F
F	T	T	T
F	F	F	F

This, combined with the simplification of b , suggests that b may be replaced with $b = d$ in the implementation, or possibly removed entirely, with reference to d in its place. ■

■ **Example 6.7** • An example of an implementation which uses the same variable in both input and output roles would be a Boolean flag, such as

`a = a || b`

The equivalent expression is $a = a \vee b$. In this case, the truth table requires two separate versions of a : one for the input and one for the output. In the framework truth table, a is listed twice: once on the left as input, and once on the right as output. Some truth tables will annotate each instance to clarify which use is intended.

 *Boolean Flag:* a Boolean variable which starts out at one value (usually false) and switches value in one direction only (usually to true) under a variety of conditions.

a	b	a
T	T	
T	F	
F	T	
F	F	

The result side of the truth table is found by filling in the evaluation of $a \vee b$ for each row. That fact that the value of a will change at the end is irrelevant.

a	b	a
T	T	T
T	F	T
F	T	T
F	F	F

■

6.6 Relationship to Natural Language

Natural language is frequently ambiguous, and so presents some issues in translating into Boolean ex-



pressions. In particular, words like “and” and “or” may have different meanings than the strict AND and OR of Boolean logic. In most cases, there is no hard and fast rule; reasoning must be employed to determine what the writer intended. If the writer is available to ask for clarification, all the better.

■ **Example 6.8** • Consider the following sign, seen at a garden store outside a showroom of antiques and fragiles: “This is a difficult room for kids and carts.” To construct a logic expression, we first find the variables: let d indicate if the room will be difficult, k indicates if we have kids with us, and c indicates if we have a cart with us. The naive approach would translate this expression into $d = k \wedge c$, but this is probably not correct. Such an expression claims that the room is difficult only if you have BOTH a kid AND a cart. It’s more likely that this “and” is actually an inclusive or, that is, the sentence is shorthand for “This is a difficult room for kids, as well as being a difficult room for carts.” The expression in this case would be $d = k \vee c$; if you have either or both, it will be a difficult room. ■

■ **Example 6.9** • A sandwich shop offers the question: “Do you want swiss or provolone cheese?”. An answer of “yes” here is probably not acceptable; we can assume this is an exclusive or; you can choose one or the other but not both. Actually, though, this isn’t quite correct because there is a third, valid choice: “no cheese”. Let v indicate if a cheese selection is valid, s means swiss is selected, and p means provolone is selected.

s	p	v
T	T	F
T	F	T
F	T	T
F	F	T

How can we represent this truth table with an expression? Either $v = \neg(s \wedge p)$, which could be read as “Any selection except for both is fine”, or $v = \neg s \vee \neg p$ (Just DeMorgan’s Law), which could be read as “Either don’t choose swiss, or don’t choose provolone, or both.” ■



Conditions provided in a bulleted list may be ANDs or ORs, or in some cases, an assortment of these. Watching for contradictions and tautologies may help to determine which case is true. In other words, if interpreting the list as ANDs yield a contradiction, try interpreting the list as ORs.

■ **Example 6.10** • Consider the following bulleted list, which describes when a daytime running light will automatically be turned on:

- The engine is running.
- The parking brake is released.
- The light switch is in the “OFF” position.
- The light switch is in the “Auto” position, but the headlights do not need to illuminate.
- The shifter is not in the park position.

At first, this list seems like a good list of AND conditions; except for the contradiction that would require the light switch to be in both the OFF and the Auto position. So these two cannot be connected with AND. The other conditions, however, certainly could and probably should be connected with AND: It wouldn’t make sense for the running lights to come on when, say the engine is off, the shifter is in park, but the parking brake has been released, as would happen if the conditions were all ORs. Besides the contradiction, however, there is no other indication of how these conditions should be combined. Thus common sense needs to be applied.

If we assume that each of these conditions is combined with AND, except for the contradictory conditions about the light switch, then we can create a logical expression that indicates d , when the daytime running light is enabled. Let e be true if the engine is running, p be true if the parking brake is applied, l be true if the light switch is in the auto position (we’ll assume auto and off are the only two positions), h be true if the headlights need to illuminate, and s be true if the shifter is in park.



In that case, $d = e \wedge \neg p \wedge (\neg l \vee (l \wedge \neg h)) \wedge \neg s$. ■



6.7 Exercises

Solutions to these exercises can be found in Appendix A.6 on page 270.

1. *Problem:* Prove that DeMorgan's Law can be extended to additional terms. In other words, prove that $\neg(a \wedge b \wedge c) = \neg a \vee \neg b \vee \neg c$.

2. *Problem:* For each of the following Boolean expressions, indicate if it is a contradiction, a tautology, or just satisfiable.

- (a) $a \vee (b \wedge \neg a)$
- (b) $\neg(\neg a \vee \neg b) \wedge \neg(a \wedge b)$
- (c) $a \vee b \vee c \vee \neg(a \wedge b \wedge c)$

3. *Problem:* Simplify the Boolean expression $(a \vee b) \wedge (a \oplus \neg b)$.

4. *Problem:* Rewrite $(a \wedge b) \rightarrow (b \rightarrow c)$ without implications, then simplify.

5. *Problem:* Prove the implication law 10b (called contrapositive), that $x \rightarrow y = \neg y \rightarrow \neg x$.

6. *Problem:* Consider the implementation

```
b = b == false
```

- (a) Convert to logical form,
- (b) Simplify, and
- (c) Describe in words

7. *Problem:* Assume response and value are Boolean variables. For this implementation:

- (a) Create a truth table, and
- (b) Simplify

```
response = false
if (value == true) then
    response = true
else
    response = false
end if
```

8. *Problem:* Assume self->active and STATUS are Boolean variables. Note that self->active is one variable, the \rightarrow does not carry any significance in terms of logical operators. For this implementation:

- (a) Create a truth table, and
- (b) Simplify

```
if (self->active != STATUS) then
    if ((self->active == false)
        && (STATUS == true)) then
        STATUS = false
    else if ((self->active == true)
        && (STATUS == false)) then
        STATUS = true
    end if
end if
```

Decision Tables



Decision Table: a logical table expressing and analyzing conditions and actions for a certain problem domain.

A decision table is a form of truth table which relaxes the requirement that every input is a two-valued Boolean, introduces the notion of impossible situations (and therefore, the lack of need to indicate them on a chart), and allows multiple output values which usually correspond to actions that a person or computer might take.

Decision tables are often used as a communication tool between programmers and domain experts (the client). A correct decision table allows both the client and programmer to be sure the correct logic will be implemented in a program. Decision tables are often represented “rotated” compared to truth tables (although this is not essential), with the rows being the input and output labels, and the columns being the various possibilities.

7.1 Boolean Decision Tables

The most basic form of a decision table is where every input and output follows the two value Boolean form.

■ **Example 7.1** • For example, let’s create a decision table that represents the following rules:

- If the order is more than \$35 and paid in cash,

apply a 5% discount.

- If the order is not paid with cash, give a membership application.

In this case, two conditions warrant our attention: whether or not the order was paid with cash, and whether or not the order was more than \$35. The possible actions are to apply a 5% discount, and give a membership application. Here is the decision table framework for these conditions and actions:

Conditions and Actions	Rules			
	1	2	3	4
More than \$35	Y	Y	N	N
Paid in cash	Y	N	Y	N
Apply 5% discount				
Give Membership App				

Each action value can be computed by considering each column of conditions against the original description. We find that the discount applies only when the order is more than \$35 and it is paid in cash. The membership application is given whenever the order is not paid with cash, regardless of the amount.

Conditions and Actions	Rules			
	1	2	3	4
More than \$35	Y	Y	N	N
Paid in cash	Y	N	Y	N
Apply 5% discount	Y	N	N	N
Give Membership App	N	Y	N	Y



A condition is an input; an action is an output.



Like a truth table, every possible situation must be considered. Thus, a standard Boolean valued decision table with n conditions will have 2^n columns.



Generally yes is associated with true, and no with false.

It is possible to construct Boolean logic expressions based on decision tables. These expressions allow the rules to be implemented directly into a computer program. One expression per action is expected. The expressions can be derived directly from the descriptive statements, or mechanically from the decision table. A mechanical transformation uses the disjunctive normal

form technique of identifying the rules in which the action is true, and building an expression based on the conditions in which the action is taken.

■ **Example 7.2** • Consider the previous example. If we let m be “more than \$35”, c be “paid in cash”, d be “apply discount”, and a be “give membership application”, then we can look at the descriptions to form the expressions $d = m \wedge c$ and $a = \neg c$.

Alternatively, we can mechanically create the expressions. In this case, $d = (m \wedge c)$ and $a = (m \wedge \neg c) \vee (\neg m \wedge \neg c)$. Normal Boolean simplification could be applied to reach the more reduced forms shown previously. ■

■ **Example 7.3** • What if the list of rules was modified to add a third rule:

- If the order is more than \$35 and paid in cash, apply a 5% discount.
- If the order is not paid with cash, give a membership application.
- Whenever a 5% discount is applied, give a membership application.

The number of business rules has little impact on the number of conditions and actions; these must be carefully inferred from reading the descriptions given. Watch out for any contradictions in the rule descriptions.



This last rule seems to add new conditions into the mix, but really it does not. What we must do is substitute in to the expression all possible cases that would result in a discount being applied. Thus, we could modify this rule to read: If the order is more than \$35 and paid in cash, give a membership application.

How does this affect our decision table? Are there new conditions? No, we still only care about the method of payment and the amount, specifically, the \$35 cutoff and the cash or not. How about actions? Is there a new action? No, there are still only two actions. Even though there are three rules given, there are only two conditions and two actions.

The modified truth table has only one change as a result of this new rule.

Conditions and Actions	Rules			
	1	2	3	4
More than \$35	Y	Y	N	N
Paid in cash	Y	N	Y	N
Apply 5% discount	Y	N	N	N
Give Membership App	Y	Y	N	Y

The logical expression for a (giving membership app) is updated to read $a = (m \wedge c) \vee \neg c$. ■

7.2 Indifferent Conditions

In many cases, a decision table with n Boolean conditions need not have strictly 2^n columns; some combinations can be simplified by indicating that a certain condition does not matter in certain cases. This makes for a decision table which is smaller and easier to understand and read.

■ Example 7.4 • Consider the following rules:

- Pay base salary to salaried employees.
- Pay hourly wage to hourly employees.
- Pay overtime to hourly employees who worked more than 40 hours.

Here the conditions are whether an employee is hourly or salaried, and whether or not they worked more than 40 hours per week. There are three possible actions: paying the base salary, the hourly wage, and/or the overtime.



Indifferent Condition: a condition whose value does not matter for certain rules.



Some people may choose to represent hourly and salaried as two different Boolean conditions. This may be useful if an employee could work a salaried position and also have additional hourly responsibilities (and pay).

Conditions and Actions	Rules			
	1	2	3	4
Salaried?	Y	Y	N	N
More than 40 hours	Y	N	Y	N
Pay base salary	Y	Y	N	N
Pay hourly wage	N	N	Y	Y
Pay overtime	N	N	Y	N





It is important to make sure that all possibilities are still covered, and that no contradictions exist, when using indifferent conditions.

One thing you may notice in this table is the similarity between rule columns 1 and 2. If a worker is salaried, their pay is not affected by the number of hours worked. Thus, “more than 40 hours” is an indifferent condition in the case of a salaried worker. We can combine rule columns 1 and 2 by marking the condition as indifferent.

Conditions and Actions	Rules		
	1	2	3
Salaried?	Y	N	N
More than 40 hours	-	Y	N
Pay base salary	Y	N	N
Pay hourly wage	N	Y	Y
Pay overtime	N	Y	N



7.3 Relationship to Boolean Expressions

Boolean decision tables, possibly with indifferent conditions, can be directly converted into Boolean expressions. One expression will be created for each action in the decision table: the expression will be true if the action should be undertaken, false otherwise. Each of the conditions will become a variable to be used in the expressions.

■ **Example 7.5** • For example, consider the decision table shown previously. There are three actions: pay base salary, which we'll call b , pay hourly wage, which we'll call h , and pay overtime, which we'll call o . The conditions are salaried, which we'll call s , and more than 40 hours, which we'll call m .

Each action is converted into an expression using a similar technique to that used for truth tables. Here are the true cases for paying the base salary:



Conditions and Actions	Rules		
	1	2	3
Salaried?	Y	N	N
More than 40 hours	-	Y	N
Pay base salary	Y	N	N
Pay hourly wage	N	Y	Y
Pay overtime	N	Y	N

An indifferent condition is simply omitted from the generated expression. Thus our expression for paying the base salary is $b = s$.

Conditions and Actions	Rules		
	1	2	3
Salaried?	Y	N	N
More than 40 hours	-	Y	N
Pay base salary	Y	N	N
Pay hourly wage	N	Y	Y
Pay overtime	N	Y	N

Two rules cause hourly wage to be paid. Within a column, combine the conditions with ANDs; between columns with ORs (disjunctive normal form). This gives us $h = (\neg s \wedge m) \vee (\neg s \wedge \neg m)$. This expression can be simplified, if desired, to $h = \neg s$.

Conditions and Actions	Rules		
	1	2	3
Salaried?	Y	N	N
More than 40 hours	-	Y	N
Pay base salary	Y	N	N
Pay hourly wage	N	Y	Y
Pay overtime	N	Y	N

Finally, paying overtime has the single rule of $o = \neg s \wedge m$. ■

7.4 Multi-valued Conditions

Decision tables also have the flexibility to replace rigid Yes/No or True/False values with multi-valued condi-



tions more appropriate to a given situation.

■ **Example 7.6** • Consider the following rules:

- Give a free gift to customers under 18.
- Give a discount card to customers 18 and up.
- Mail a coupon to customers over 45.

Here we have three actions: free gift, discount card, and coupon. But what are the conditions? There is one main variable being considered: age. However, it is not a simple yes/no cutoff. There appears to be several different cutoffs. One approach is to break these into three different conditions: “under 18?”, “between 18 and 45?”, and “over 45?”. This approach is clunky and awkward.

A better alternative is to allow conditions to have more than two values. In this case, instead of yes and no, we can have values < 18 , $18 - 45$, > 45 .

The total number of columns in a decision table with no indifferent conditions can be determined using the multiplicative counting rule, that is, the product of the number of options for each condition.

Conditions and Actions	Rules		
	1	2	3
Age	< 18	$18 - 45$	> 45
Free gift	Y	N	N
Discount card	N	Y	Y
Mail coupon	N	N	Y

If we add in conditions relating to customer membership, allowing a customer to be either a non-member, a basic member, or a premier member, what happens to the column count? Assuming no indifferent conditions, there would be nine columns. Every permutation of possibilities must be considered. In this example the actions would be filled in the usual way, based on specified rules.



For reasons of space, premier is abbreviated Pr in this table.

Conds. and Actions	Rules								
	1	2	3	4	5	6	7	8	9
Age	< 18	< 18	< 18	18 – 45	18 – 45	18 – 45	> 45	> 45	> 45
Membership	Non	Basic	Pr	Non	Basic	Pr	Non	Basic	Pr
<i>Actions</i>	...								

When translating multi-valued conditions into Boolean expressions, the multi-valued conditions must first be decomposed into a set of Yes/No conditions. ■

7.5 Impossible Conditions

Certain combinations of conditions may be physically or conceptually impossible.

■ Example 7.7 • Consider the following rules:

- Offer a credit application to male customers.
- Offer a gift bag to pregnant customers.
- Offer a coupon to female customers.

Creating a decision table in the standard way produces:

Conditions and Actions	Rules			
	1	2	3	4
Gender	M	M	F	F
Pregnant?	Y	N	Y	N
Credit app	Y	Y	N	N
Gift bag	Y	N	Y	N
Coupon	N	N	Y	Y

If needed for conversion to Boolean expressions, multi-valued conditions can be decomposed into a series of Boolean conditions. There would be one condition for each value, and, using impossible conditions, each rule would have exactly one of those values be true.

Something is wrong with this table. In general, it is not possible for a customer to be both male and pregnant. The client may intend to include men whose partners are pregnant, men who are adopting, or other circumstances. However, if so, this should be clarified and the



An indifferent condition is similar to an impossible condition; however, an indifferent condition indicates that both cases are possible, but there is no difference between them. An impossible condition indicates a particular case cannot occur.



Undefined Behavior: actions may or may not occur when the conditions are not specified by a decision table.



conditions updated to more accurately reflect testable facts. If, however, only pregnant women are intended, then the condition of pregnant men can be removed from the table.

Conditions and Actions	Rules		
	1	2	3
Gender	M	F	F
Pregnant?	N	Y	N
Credit app	Y	N	N
Gift bag	N	Y	N
Coupon	N	Y	Y

This decision table, although it appears incomplete, is still acceptable because all possible conditions have been considered. In this case, if somehow a pregnant male were found, the system's behavior is undefined. Noting impossible conditions is important, as in the future, situations may change that allow formerly impossible conditions to become possible, and actions must then be defined. ■

Undefined behavior is very dangerous for programmers if conditions that turn out to be possible were previously thought to be impossible, or missed. In that case, it is unknown what, if any, actions will be executed by the program. In particular, there is no guarantee that an error message will be generated, which may lead to silent failures.



7.6 Exercises

Solutions to these exercises can be found in Appendix A.7 on page 277.

1. *Problem:* Consider selecting an appropriate college to attend. The student posits the following requirements:

- College must be in the Northwest region, unless it is a top ten college.
- College must offer computer science, or math, or both.
- Computer science program, if offered, must have at least one renowned faculty member.

Create a decision table to indicate if a college meets the requirements. Convert the decision table into Boolean expressions.

2. *Problem:* Decompose the following multi-valued conditions into Boolean conditions. Convert the resulting decision table into a Boolean expression.

Conds. and Actions	Rules								
	1	2	3	4	5	6	7	8	9
Age	< 18	< 18	< 18	18 – 45	18 – 45	18 – 45	> 45	> 45	> 45
Membership	Non	Basic	Pr	Non	Basic	Pr	Non	Basic	Pr
Call from Trainer	N	N	N	N	Y	Y	Y	Y	Y

3. *Problem:* Create a decision table for the following statement:

A mailing is to be sent out to customers. The content of the mailing is about the current level of discounting and potential levels of discounting. The content is different for different types of customers. Customer Types A, B, and C get a normal letter except Customer Type C, who get a special letter. Any customer with 2 or more current lines or with a credit rating of 'X' gets a special paragraph added with an offer to subscribe to another level of discounting.

Convert the resulting decision table into a Boolean expressions.

4. *Problem:* Create a decision table for the following statement:

If the package weight is less than 5 pounds, base shipping is \$4.00. If the package weight is 5 pounds to 10 pounds, base shipping is \$6.00. For packages more than 10 pounds, base shipping is \$10.00. If overnight shipping is selected, add \$20.00 to the shipping cost. If insurance is selected, double the base shipping price. Insurance is mandatory when overnight shipping is used. Packages 5 pounds or more should have a "heavy" label applied. If insurance is selected or the package is more than 10 pounds, have a "special freight" label applied.

5. *Problem:* Consider the following decision table:

Conditions and Actions	Rules		
	1	2	3
Enrolled in Class	Y	Y	N
Passed Most Recent Exam	Y	-	N
Advertise Next Class	Y	N	Y
Send Checkup Email	Y	Y	N

- (a) Indicate all conditions which have contradictory actions.
- (b) Indicate all conditions which are undefined.
- (c) For all undefined conditions, is the condition probably impossible or simply undefined?

Consider the following decision table (called the original table) below:

Conditions and Actions	Rules							
	1	2	3	4	5	6	7	8
Under \$50	Y	Y	Y	Y	N	N	N	N
Pays by check	Y	Y	N	N	Y	Y	N	N
Pays by credit card	Y	N	Y	N	Y	N	Y	N
Call Supervisor	N	N	N	N	Y	Y	N	N
Check Photo ID	Y	Y	Y	N	Y	Y	Y	N
Proceed with sale	Y	Y	Y	Y	N	N	Y	Y

Assume that each purchase can be paid by only one method of payment: cash, check, or credit card.

- 6. *Problem:* Simplify the original table by applying indifferent conditions.
- 7. *Problem:* Simplify the original table by detecting impossible conditions.
- 8. *Problem:* Simplify the original table by applying multi-valued conditions.

Logic Circuits

Concepts of Boolean algebra remained a fringe science for years until the development of the electronic computer. With the advent of electric switches first as vacuum tubes, and today as transistors, operations such as AND, OR, and NOT could be implemented in a physical form. Indeed, these operations form the foundation of computing as we know it; all the advanced capabilities of your computer boil down to an endless sea (millions or more) of tiny microscopic switches implementing logical expressions.

A logic circuit diagram shows how various operations are connected together. Rather than relying on order of operation and parentheses, as in a written expression, a circuit diagram uses lines or “wires” to connect the operations. The result of a logical circuit can be found by tracing a true or false value from operation to operation (along the wires) until it arrives at the end of the circuit. Even if physical implementation is not our goal, logic circuits provide a helpful conceptual framework for visualizing logical expressions.

 *Logic Circuit:* Boolean operations, indicated with specific symbols, connected with wires which show order of operation.

8.1 Fundamental Logic Gates

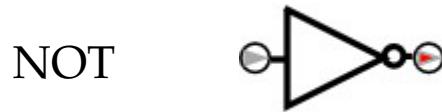
Each Boolean operation is visualized with a specific symbol, called a gate. Most gates have one or two in-

puts and one output; these correspond to whether the operation is a binary operation (AND, OR, XOR, etc.) or a unary operation (NOT). The output of a given gate based on its inputs is usually defined by the appropriate truth table.

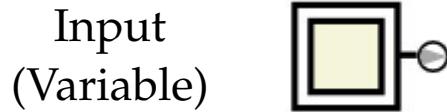


The exact shape of gates may vary from one system to another; the arrows are used here only for clarification and are not an essential part of the gate's representation.

The symbols for the fundamental gates are shown below. Note that the inputs arrive on the left, and the output is produced on the right. However, there is no reason this directionality is essential. The gates can be rotated or flipped in any way: the flatter side will be the “input” side and the pointy side will be the “output” side.



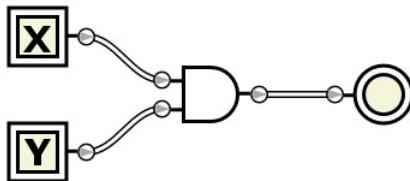
In addition, for the purposes of this text, inputs (variables) will be defined with a square box, and the output will be defined with a circle.



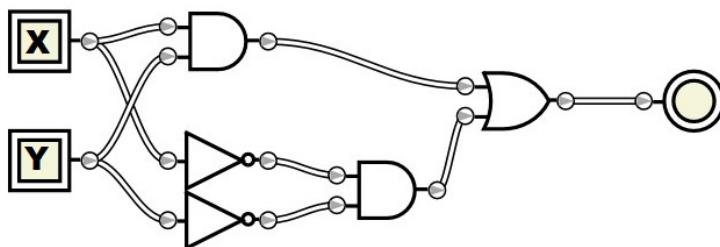
8.2 Logic Circuits

A logical expression is specified by identifying each variable, which becomes an input, and then connecting, with wires, various gates to form the same order of operations.

■ **Example 8.1** • For example, consider the expression $x \wedge y$. This expression has two variables, so there will be two inputs. These inputs are the start of the flow of true/false through the circuit. From the inputs, the circuit will enter an AND gate, and the result thereof will flow to the output.



■ **Example 8.2** • Consider the expression $(x \wedge y) \vee (\neg x \wedge \neg y)$. Even though this expression is more complex, it still has two distinct variables, so it will have two inputs. These inputs will be combined to form $x \wedge y$, and separately, they will be inverted to find $\neg x$ and $\neg y$ respectively. These two values will be combined in an AND, and finally an OR will complete the circuit.



It is possible to determine the result of a particular configuration of inputs by tracing through the circuit

Logic circuits in this book are produced using the Logic Gate Simulator software found at <http://www.kolls.net/gatesim>



A logic gate's output may be used many times (there may be many wires connected to an output), but a particular logic gate's input may only have one source.

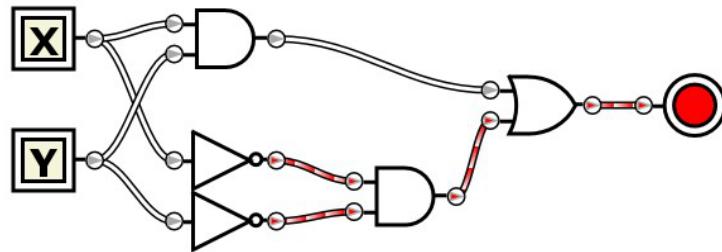


The software used to generate these circuits can show the state (true/false) of all parts of the circuit by coloring wires that have a "true" value. This coloration may change depending on various inputs and is not an essential part of the diagram.



from gate to gate. Light color means false, darker color means true.

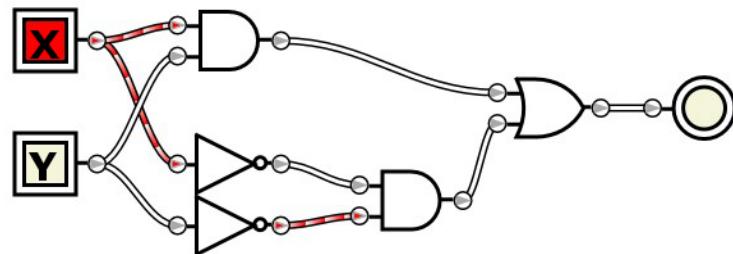
■ **Example 8.3** • Here is the state of the circuit if both inputs are false. The circuit output is true in this case.



Circuits are evaluated strictly by following wires from gate to gate; the placement of the gates within the circuit is not significant.



The previous diagram shows that if x is false and y is also false, the output of the circuit is true. The same circuit is now colored to show x being true and y being false. This causes the circuit output to be false.



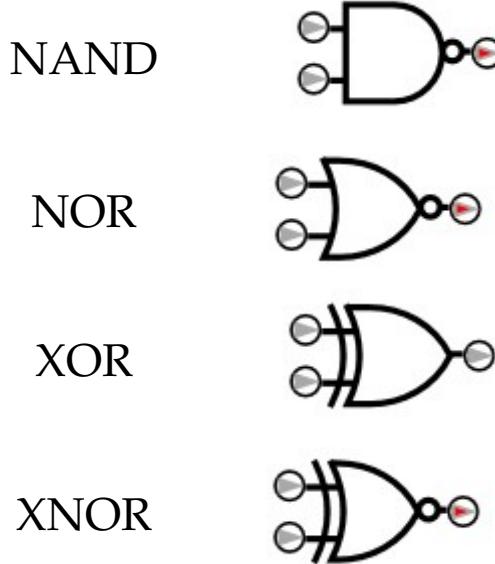
Each of these configurations correspond to a row in the circuit's truth table. It is possible to determine the entire truth table for a circuit using this technique, one input permutation at a time.

8.3 Compound and Universal Gates

Four common compound gates exist. These gates are so-called because they can be thought of as the con-

nection of several other basic gates. In physical construction, compound gates may not actually be made of the basic gates, but they will have the same behavior. In particular, NAND gates are cheap to make and sufficient to represent any circuit, so many circuits are constructed entirely of NAND gates.

 The little circle at the output edge of the gate means "followed by NOT". Be very careful to notice when this circle is present and absent; it makes a big difference!



The NAND gate is AND followed by NOT. The NOR gate is OR followed by NOT. The XOR gate is exclusive-or: one or the other, but not both. This is represented in Boolean expression as $x \oplus y$. The standard definition of \oplus can also be used to expand the XOR gate into a circuit of basic gates.

Finally, the XNOR gate is exclusive-or followed by NOT. Remember that XOR is true when either but not both of the inputs are true. Therefore, XNOR is true when neither or both inputs are true. This is the same as equality in Boolean expressions: $x \leftrightarrow y$.

NAND and NOR have a unique status as universal logic gates. A sequence of NAND or NOR gates can



Universal Logic Gate: a gate that can implement any other logic gate or circuit.

be used to produce any other logic gate or circuit. Therefore, it is often desirable to convert a circuit into one consisting entirely of NAND or NOR gates. Even though the gate count will be higher, production is simplified by manufacturing only a single type of gate.

In order to be universal, a gate must possess two properties: it must have inversion (the ability to make a false into a true, and vice versa), and it must have selection (the ability to distinguish in its input true from false). The gates AND, OR, and XOR lack inversion (you cannot get a true out of these gates by feeding only false values). The gates XNOR and NOT lack selection (XNOR can only tell you if the inputs are equal, but not if they are true or false; NOT has only one input and so can't have two or more different values at once). This leaves only NAND and NOR from the gates we have seen that are universal.

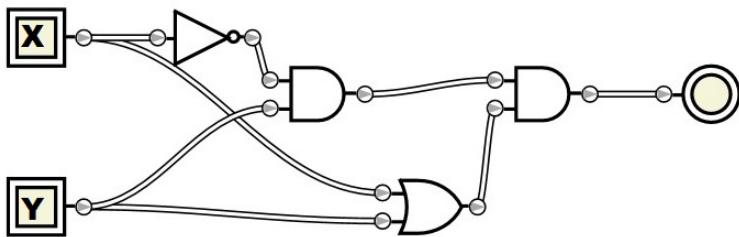
8.4 Converting to Boolean Expressions

Every linear logic circuit (that is, a logic circuit with no loops) can be converted into a Boolean expression, and every Boolean expression can be converted into a logic circuit. Like Boolean expressions, the only way to guarantee or disprove that two logic circuits are equivalent is to create and compare truth tables for both. Simplification of circuits using the Boolean algebra rules is also a good technique.

A logic circuit can be converted to a Boolean expression by building up subexpressions one gate at a time. Start at the inputs to the circuit, and follow the wires to each gate. Apply the Boolean operator of that gate to the predetermined values of the inputs.

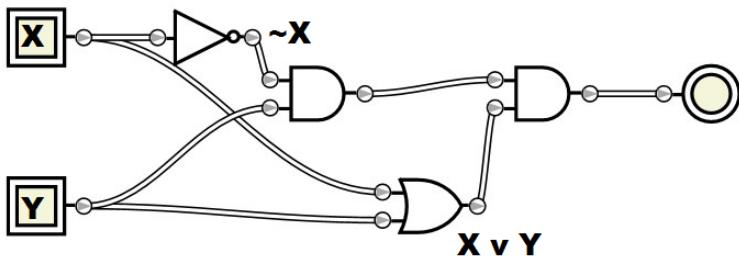
- **Example 8.4** • For example, consider the following circuit:



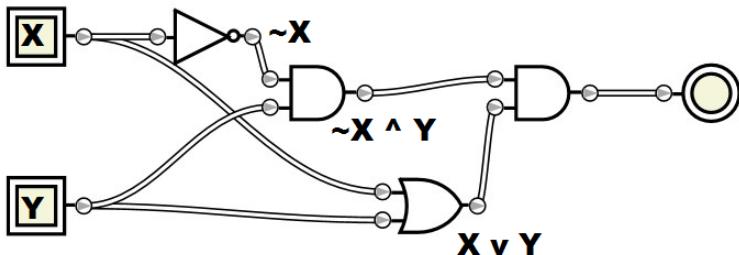


Follow the wires from the inputs on the left to the first gates encountered: the NOT gate on top and the OR gate on the bottom. Write subexpressions for these gates next to the gates themselves. These subexpressions will form the input to subsequent gates.

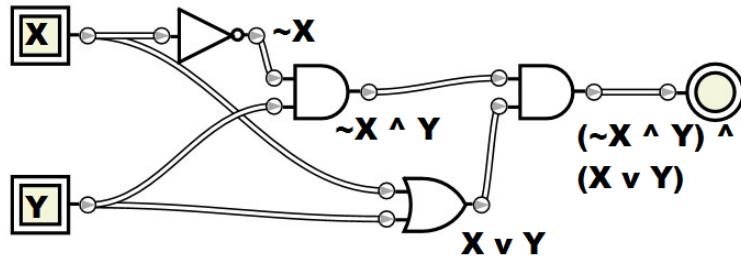
Again: Circuits are evaluated strictly by following wires from gate to gate; the placement of the gates within the circuit is not significant. The presence of the AND gate “before” the OR gate from left to right is not significant. The ordering is determined by wire sequence only.



Next, we can proceed to the AND gate in the middle. The AND gate places the logical AND operator between the two inputs, which we take from the parts of the circuit annotated.



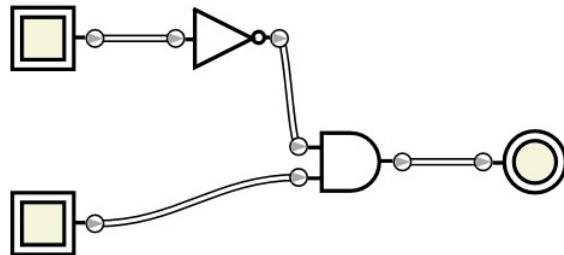
Finally, the last AND gate combines the two subexpressions we have calculated. Be sure to use parentheses to enforce order of operations!



The equivalent Boolean expression for this circuit is $(\neg x \wedge y) \wedge (x \vee y)$. Normal techniques can be used to write a truth table for this expression (which would also be the truth table for the circuit):

x	y	$(\neg x \wedge y) \wedge (x \vee y)$
T	T	F
T	F	F
F	T	T
F	F	F

By observation from the truth table, or by Boolean algebra laws, the expression can be simplified to $\neg x \wedge y$, which means the circuit can also be simplified:



■

8.5 Converting from Boolean Expressions

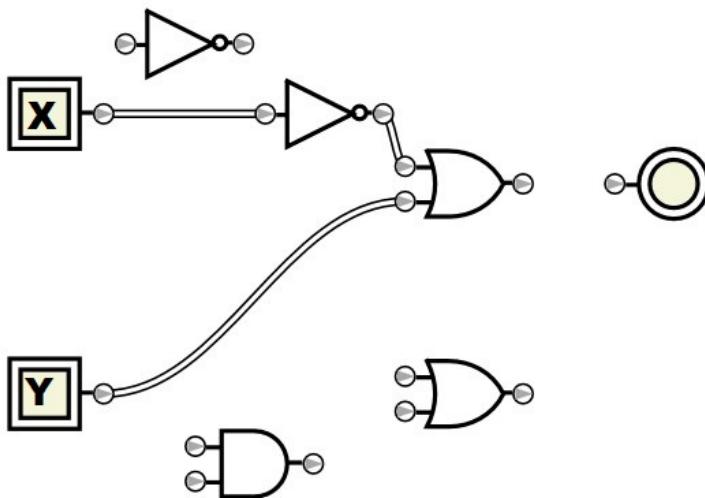
Given a Boolean expression, it is also possible to create an equivalent logic circuit. Each operator in the expression will be represented by a gate in the circuit. The



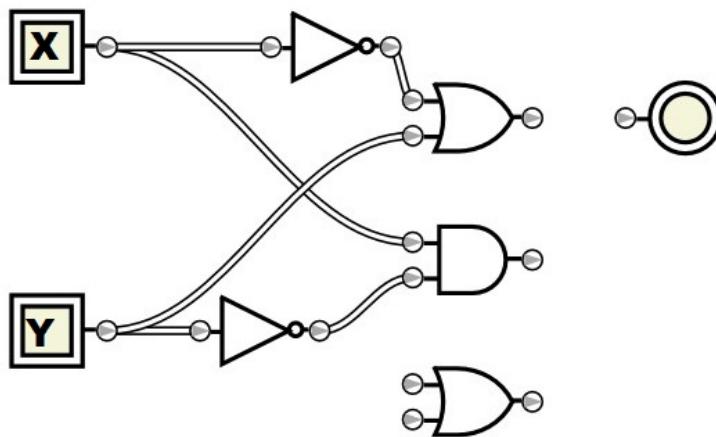
wiring process is inside-out: start at the innermost operation, and build outward. Alternatively, you could start at the output and build the circuit outside-in.

■ **Example 8.5** • For example, consider the expression $(\neg x \vee y) \vee (x \wedge \neg y)$. By counting the operators, we can tell the circuit will have two OR gates, two NOT gates, and one AND gate. Using the inside-out approach, we find there are two parentheses blocks. Entering the left block, we know that the NOT binds strongest.

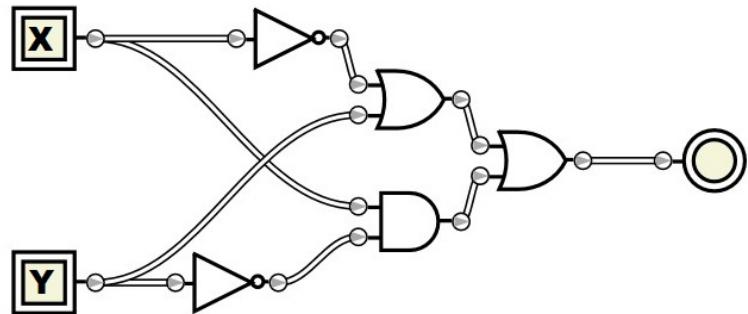
In this example, I place all gates onto the canvas first, and then connect them. Unconnected gates serve no inherent purpose, and you could omit the gates until you needed them.



Next, the other parentheses block is constructed similarly. Note that it uses the same x and y input values. This reuse of values is not a problem.

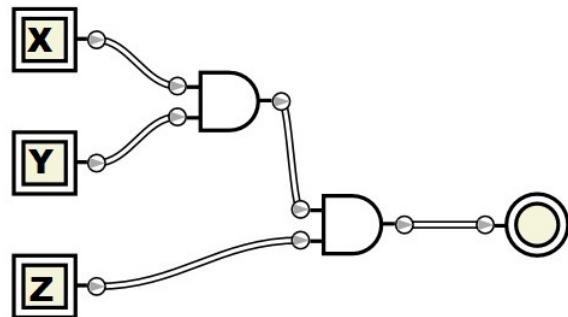


Finally, the output of these two subexpressions are connected with the OR gate, which provides the circuit output (expression result).



■

■ **Example 8.6** • Consider the Boolean expression $x \wedge y \wedge z$. How could this expression be represented using a logic circuit? A straightforward approach as described previously would create two AND gates: one for $x \wedge y$ and the other for the output of that expression AND with z . The circuit would look like:

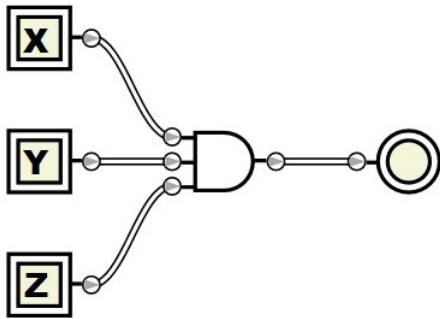


Multi-input NAND and NOR gates still have only one NOT operation at the end. Therefore, a multi-input NAND is not like several NANDs in sequence: it is like several ANDs in sequence, followed by a single NOT.



An acceptable shorthand is to use a single AND gate with more than two inputs. Binary AND, OR, NAND, and NOR gates have a convention that allows multiple inputs. These inputs extend the logical meaning of the gate, in the same way that a sequence would. For example, here is the same gate shown with a single three input AND gate:



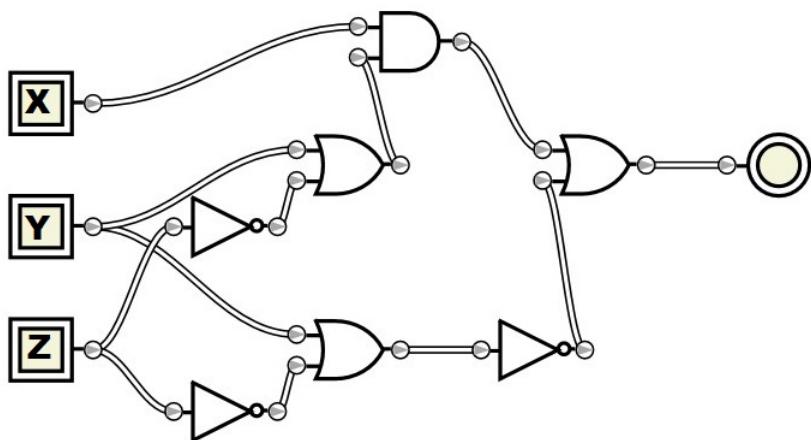


A three input NOR with variables x , y , and z would have the expression $\neg(x \vee y \vee z)$.

8.6 Partial Circuit Reuse

Circuits provide an opportunity, not exposed by Boolean expressions, for taking advantage of reuse. Reuse (after, perhaps, abstraction) is one of the most important skills for a computer programmer, and logic circuits provide an intuitive field for working with concepts of reuse.

■ **Example 8.7** • Consider the Boolean expression $(x \wedge (y \vee \neg z)) \vee \neg(y \vee \neg z)$. First consider a naive implementation of this expression into a circuit:



In the usual way, this circuit has one gate for each logical operator in the expression. The wiring sequence

 *Reuse*: using a single block of code, expression, or part of a circuit for several purposes. Also referred to as “DRY”, short for “Don’t Repeat Yourself”.

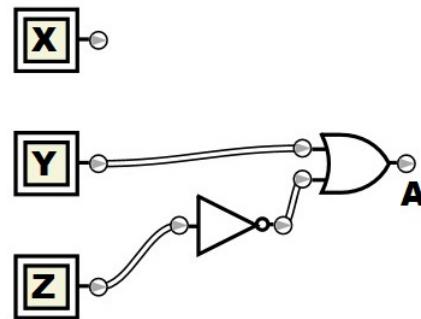
follows the “inside out” order of operations, so this circuit is directly based on the expression.

Without simplifying the expression, however, we could still simplify the circuit. Note that part of this expression is repeated: $y \vee \neg z$. Just as the values of y and z are used multiple times in the circuit, the values of $y \vee \neg z$ could be used multiple times. For simplicity, we’ll let $a = y \vee \neg z$. This subexpression can then, in a sense, be considered like an input value into an expression. We can rewrite the expression to take advantage of the definition of a , and provide a new circuit which provides the value of a as defined.

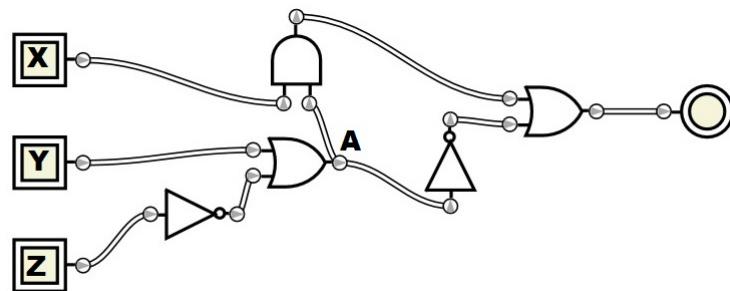
It’s important to ensure that such a replacement is valid under order of operations. For example, if the expression included a portion like $x \wedge y \vee \neg z$, without parentheses, transforming that expression to $x \wedge a$ would NOT be valid due to order of operations being violated.



The sideways representation of several gates in this diagram is for layout purposes. The rotation of a gate has no impact on its operation.



The new expression would be $(x \wedge a) \vee \neg a$. This expression was found by simply replacing all references to $(y \vee \neg z)$ in the original expression with a .



The abbreviated circuit now uses only five gates, compared to the original’s seven. This reduction is accomplished by reusing portions of the circuit which were duplicated. ■



8.7 Disjunctive Normal Form

As discussed previously, a standard form for Boolean expression is disjunctive normal form. This form is verbose, though often the initial form of an expression if it is derived from a truth table. The same form can also be applied to logic circuits. This form is designed to make circuit analysis more straightforward.

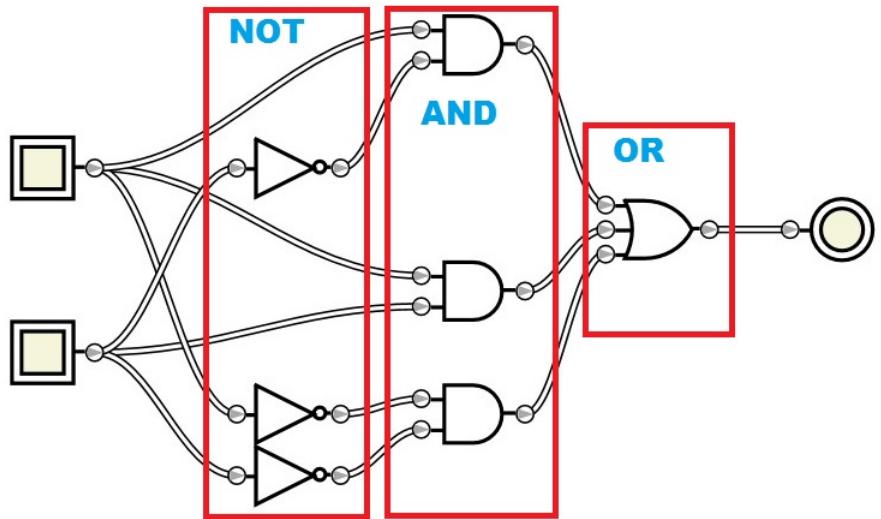
A logic circuit is in disjunctive normal form if:

- The circuit is entirely feed forward from inputs to output. (That is, there are no loops in the circuit)
- Any NOT gates occur first, using only the original inputs.
- Any AND gates occur next, using only the original inputs and output from the NOT gates.
- The result of all the AND gates are combined in an OR gate,
- which connects to the output.

A Boolean expression which is in disjunctive normal form, when translated into a logic circuit, will yield a logic circuit in disjunctive normal form. It is also possible to convert any feed forward logic circuit into disjunctive normal form. This would normally be accomplished by finding the truth table for the circuit, and then reconstructing from the truth table the disjunctive normal form expression.

■ **Example 8.8** • Consider the expression $(x \wedge \neg y) \vee (x \wedge y) \vee (\neg x \wedge \neg y)$. This expression is in disjunctive normal form.





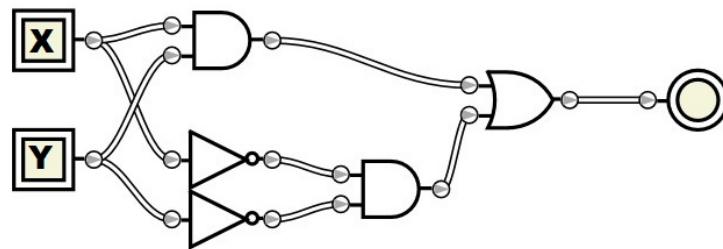
■

Circuits in disjunctive normal form may be simplified in many cases. In some cases, a simpler version of the circuit still in disjunctive normal form is possible, as long as the sequence rules are followed.

8.8 Exercises

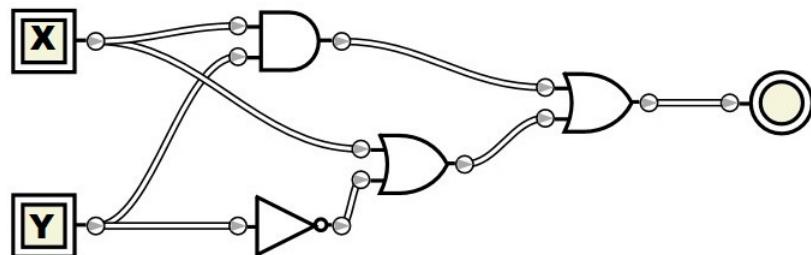
Solutions to these exercises can be found in Appendix A.8 on page 284.

1. *Problem:* Determine the output of the given circuit if x is false and y is true.

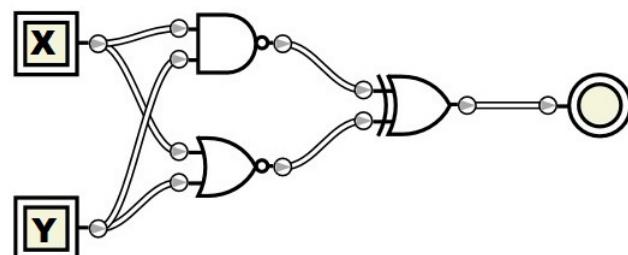


2. *Problem:* Show the definition of exclusive or, \oplus , in a circuit using only basic gates.

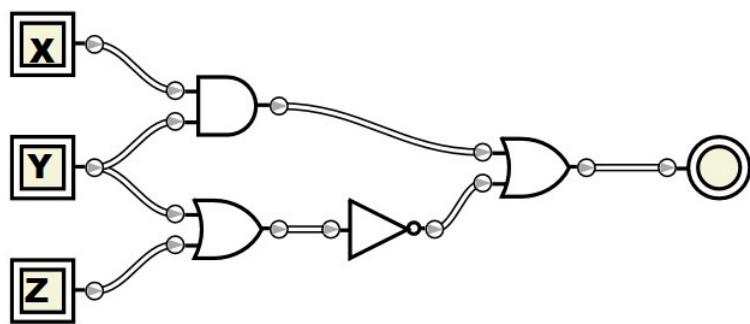
3. *Problem:* Create a Boolean expression for the circuit shown.



4. *Problem:* Create a Boolean expression for the circuit shown.



5. *Problem:* Prove that NAND is universal by using NAND gates to construct circuits which simulate:
- an AND gate
 - an OR gate
 - a NOT gate
6. *Problem:* Create a logic circuit for the expression $\neg(\neg a \vee (a \wedge b))$.
7. *Problem:* Create a logic circuit for the expression $\neg(\neg a \vee \neg b) \vee (b \wedge (\neg a \vee \neg b))$. Take advantage of reuse to avoid unnecessary gates.
8. *Problem:* Convert this circuit into disjunctive normal form.



PART III

Numbers and Letters

9 Number Systems	90	10.7 Exercises	111
9.1 Bases and Binary	91		
9.2 Octal and Hexadecimal	92		
9.3 Converting to Decimal	94		
9.4 Converting from Decimal	96		
9.5 Unary	98		
9.6 Exercises	99		
10 Integer Numbers	100		
10.1 Unsigned Integers	101		
10.2 Unsigned Addition	101		
10.3 Signed Integers	102		
10.4 Signed Addition and Subtraction	105		
10.5 Binary Multiplication	107		
10.6 Binary Coded Decimal	108		
11 Floating Point Numbers	112		
11.1 Scientific Notation	112		
11.2 Binary Number Composition	114		
11.3 Special Cases	117		
11.4 Floating Point Errors	118		
11.5 Decimal Floating Point	119		
11.6 Exercises	122		
12 Unicode and ASCII	123		
12.1 ASCII	123		
12.2 Unicode	125		
12.3 Display Issues	127		
12.4 Exercises	129		

Chapter **9**

Number Systems



Number System: a notation for representing numbers using symbols.

In previous chapters, we have dealt with collections of values, with an emphasis on true/false values. Now we will move on to look at how these values can be put together to form more complicated information. The most important, and most fundamental, type of information represented in a computer is a number. In order to understand how computers store and manipulate numbers, some fundamental concepts of numbering must first be considered.

Most modern humans represent numbers using the Arabic decimal system. This system is positional, which means that the location of a digit (the “place”) determines the value of the digit. There are ten symbols, called digits, each which represents a specific value. In addition, the Arabic decimal system includes the concept of zero, and has a symbol (digit) to represent it.

The most unusual number system in wide use is Roman numerals. Roman numerals are neither positional (X means 10 no matter where in the sequence it appears), nor do they generally have (or need) a symbol for zero. Tally marks are another form of number system. Tally marks use only one symbol (or two; if you allow a five block to be a separate symbol) and do not have a symbol for zero.

9.1 Bases and Binary

Most computer number representations are based on positional systems, with zero, like the decimal system. However, because computers are fundamentally based on true/false (two values) rather than counting with fingers (ten values), the number systems used by computers differ from the usual decimal system used by people.

The number systems significant for our purposes can be differentiated by bases. The base of a number system is how many different symbols (digits) are available in that system. As usual, we start with 0 and count up until we reach the highest digit. Then we return to 0, add 1 to the next place over, and continue. The decimal system is base 10 because there are ten unique symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). If we want to count past 9, the next number involves using two existing symbols instead of a further unique symbol: 10.

Computers use only two symbols internally: true and false. These are represented not as actual symbols, but as different electric currents in a circuit. The values of these symbols are numerically defined as 1 for true, and 0 for false. There are two symbols, so this number system is base 2, and is called binary. A symbol in binary (either 0 and 1) is also called a bit, short for binary digit.

Counting in binary works the same as counting in decimal, except that there are fewer symbols and so more places are needed more quickly. Here are the binary numbers zero through ten: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010. In order to interpret these values, the appropriate place values must be used. Decimal has tens, hundreds, thousands, and so on. Mathematically, the places in decimal (from right to left in a number) are $10^0, 10^1, 10^2$, and so on. Notice the reoccurring 10 here, due to decimal's base 10. In binary, the place values are $2^0, 2^1, 2^2$, and so on.

With multiple possible number bases in use, the simple appearance of digits is no longer sufficient to define a number's value. What is 11? Is it eleven (base 10 inter-



Base: number of symbols or digits in a particular number system.



The word decimal has root deci, meaning ten.



Binary: base 2 number system with digits 0 and 1.



Bit: binary digit, a single number with the value 0 or 1.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

For any base n , where $n > 1$, the place values from right to left are n^0, n^1, n^2 , and so on.



Numbers with bases are usually read as a sequence of digits followed by their base. For example, 101_2 would be read “one zero one base two”.



For any base n , where $n > 1$, $10_n = n_{10}$



Octal: base 8 number systems with digits 0 through 7.



Hexadecimal: base 16 number systems with digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.



Hexadecimal is often shortened to “hex”.



Like in decimal, other bases may have 0's added or removed to the front of a number without changing its value.

interpretation), or is it three (base 2 interpretation)? In order to clarify the interpretation of any number when multiple bases are in use, the base itself may be attached as a subscript to the end of the number. For example, 11_{10} would be eleven, while 11_2 would be three.

9.2 Octal and Hexadecimal

Although computers use binary internally, binary numbers quickly become very large and awkward. Therefore, two other bases are often used, which have the advantage of being directly translatable to and from binary.

Octal, base 8, directly represents three bits; Hexadecimal, base 16, directly represents four bits. If we count in binary up to three bits, we get 0, 1, 10, 11, 100, 101, 110, 111. There are eight values in that list (zero through seven in decimal). By using base 8, we can represent any three bit block with the symbols 0 through 7. Hexadecimal works the same way with four bits: there are sixteen different binary numbers of up to four bits.

Hexadecimal does have one additional challenge: in other bases, we generally use the same symbols as the decimal system. However, the decimal system provides only ten symbols, and hexadecimal requires sixteen. The six additional symbols are taken from the English alphabet.

■ **Example 9.1** • Consider the number 110101000010_2 . We can easily represent this number in octal (base 8) by breaking it into three bit chunks: 110 101 000 010. Then, each three bit chunk is replaced by the appropriate single octal digit, found by counting. 110_2 , for example, corresponds to 6_8 . The octal number is 6502_8 .

■ **Example 9.2** • Converting from octal to binary is the same process in reverse: replace each octal digit with the three corresponding bits. The octal number 107_8 has the bit blocks 001 000 111, forming the binary number 1000111_2 . Notice that the leading zeros were re-

moved, as is common unless a fixed bit width is specified. ■

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

■ **Example 9.3** • Consider the same number 110101000010_2 . We can easily represent this number in hexadecimal (base 16) by breaking it into four bit chunks: 1101 0100 0010. Then, each four bit chunk is replaced by the appropriate single hexadecimal digit. If the value is greater than 9, then a letter symbol must be used. In this case, the leftmost block $1101_2 = 13_{10}$. We need a single symbol in hexadecimal to represent the value 13. Which symbol? The convention is that $A_{16} = 10_{10}, B_{16} = 11_{10}$ and so on. So we find $1101_2 = D_{16}$. The complete conversion is $D42_{16}$. ■

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111



In computer programs, it is uncommon to represent binary numbers directly due to their length. Instead, hexadecimal or occasionally octal are used as a form of “short-hand” for binary. However, most programming systems do not have the capability to use subscripts to indicate the base of a number, so another convention is used instead. Numbers that begin with the digit 0 may be considered to be octal numbers, that is, a number like 0537 would be interpreted as 537_8 . Numbers that begin with 0x may be considered to be hexadecimal numbers, that is, a number like 0x537 would be interpreted as 537_{16} . Although the digits in these two numbers are the same, they have different values.

Many websites, and usually the operating system’s included calculator, will perform a variety of base conversions quickly and easily.



For any n , $n^0 = 1$, so often just the digit value is written without a multiplication.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

9.3 Converting to Decimal

Any number using a system like one we have seen here can be converted to decimal by multiplying each digit by its place value and adding the results. The place values are found using the numbers base. For any number abc_n where a , b , and c are digits, and n is the base, this number can be converted to decimal by taking the place values (from right to left) n^0 , n^1 , n^2 (and so on, if there were more digits), and multiplying each place value by the respective digit, so $c * n^0$, $b * n^1$, and $a * n^2$. The decimal equivalent is the sum of these terms: $a * n^2 + b * n^1 + c * n^0$.

To make the conversion easier, it may be helpful to lay out a conversion table consisting of each digit and the digit’s place values. Such tables will be shown in the examples.

■ **Example 9.4** • To take a concrete example, let’s convert $56A3_{16}$ into decimal. Lay out each digit, and below it put the place value. Recall the place value is the base of the number to the power of the digit position, with zero being the rightmost digit.

$$\begin{array}{ccccccc} 5 & 6 & A & 3 \\ 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$



In the case of hexadecimal, we may need to substitute in the value for any digit in the A through F range.

$$\begin{array}{r} 5 \quad 6 \quad 10 \quad 3 \\ 16^3 \quad 16^2 \quad 16^1 \quad 16^0 \end{array}$$



Recall that $n^0 = 1$ and $n^1 = n$.

Using each place value, we can calculate the value of each place, and then sum them.

$$\begin{array}{rcl} 5 * 16^3 & = & 20480 \\ 6 * 16^2 & = & 1536 \\ 10 * 16^1 & = & 160 \\ 3 * 16^0 & = & 3 \\ \hline & & 22179 \end{array}$$

If a_n is a single digit number, then $a_n = a_{10}$. Substitution of the value for hexadecimal digits may be required.

Adding the results gives the value 22179, so we can say $56A3_{16} = 22179_{10}$. The identical procedure (adjusted for appropriate place values) suffices for numbers of any length and of any base. ■

If the number to convert has a fractional component, the place values continue using negative powers.

■ **Example 9.5** • For example, consider converting 101.01_2 into decimal.

$$\begin{array}{r} 1 \quad 0 \quad 1 \quad 0 \quad 1 \\ 2^2 \quad 2^1 \quad 2^0 \quad 2^{-1} \quad 2^{-2} \end{array}$$

$$\sum_{n=1}^{\infty} \frac{a_n}{n^b} \quad a^{-b} = \frac{1}{a^b}$$

Calculate the place values. Note that the decimal point is useful only in determining which powers the place values have; it is not carried into the conversion, as the negative exponents will automatically “do the right thing”.

$$\begin{array}{rcl} 1 * 2^2 & = & 4 \\ 0 * 2^1 & = & 0 \\ 1 * 2^0 & = & 1 \\ 0 * 2^{-1} & = & 0 \\ 1 * 2^{-2} & = & 0.25 \\ \hline & & 5.25 \end{array}$$



Adding the terms tells us that $101.01_2 = 5.25_{10}$. ■

An alternative technique for converting a number to decimal takes advantage of an algebraic property of the previously described conversion. In the alternative technique, the sum is computed incrementally rather than all at once. Start with the left most digit. As long as more digits remain, multiply the current result by the base and add the next digit.

For example, to convert 521_8 into decimal, we start with the sum being 5. More digits exist, so multiply the current sum by the base $5 * 8 = 40$ and add the next digit $40 + 2 = 42$. Another digit exists, so multiply the current sum by the base $42 * 8 = 336$ and add the next digit $336 + 1 = 337$. No more digits remain, so this is the final answer. $521_8 = 337_{10}$.

For any two numbers with the same digits but different bases, call them a_b and a_c , if $b < c$ then $a_b < a_c$.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

If using a calculator, you can find the remainder of any division $a \div b$ by first dropping the fractional part to find the whole number quotient, represented as $\lfloor a \div b \rfloor$. Multiply this whole number result by b and subtract from a with the formula $a - b * \lfloor a \div b \rfloor$. For the example of $506 \div 8$, the calculator shows 63.25. The whole number quotient is just 63. $506 - 8 * 63 = 2$, so the remainder is two.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

9.4 Converting from Decimal

Any decimal number can be converted to another base using a process of repeated divisions. For these divisions the remainder must be recorded. Most calculators do not show the remainder natively, so be sure to find the remainder and not just use the fractional portion.

Converting a decimal whole number to another base begins by dividing the decimal number by the base. Save the quotient (whole number result) for the next cycle, and the remainder of the division is the digit. The first division produces the rightmost digit. The quotient (result) of the first division then becomes the next decimal whole number to divide, and the process repeats until the quotient is zero.

■ **Example 9.6** • For example, consider the number 506_{10} . Convert this number into octal (base 8). First, find $506 \div 8$. This gives a quotient of 63 with a remainder of 2. Repeat the process with $63 \div 8$ which gives a quotient of 7, remainder 7. Finally, $7 \div 8$ gives a quotient of 0 with remainder 7. The process stops at this point because the last quotient was 0. The resulting



number is found by assembling the remainders, with the first remainder being the rightmost digit. Therefore, $506_{10} = 772_8$.

$$\begin{array}{rcl} 506 \div 8 & = & 63r2 \\ 63 \div 8 & = & 7r7 \\ 7 \div 8 & = & 0r7 \end{array}$$

■

■ **Example 9.7** • When a base larger than 10 is used, the appropriate digit symbols must be substituted for any remainder values 10 or larger. For example, consider the number 254_{10} . Convert this number into hexadecimal (base 16). In this case, the division $254 \div 16 = 15r14$. Next, we take the quotient result and repeat the division: $15 \div 16 = 0r15$. The two remainders, with the first being the rightmost, are 14 and 15. Each of these should correspond to one symbol, so the appropriate answer is FE_{16} . In this case, 1514_{16} would *not* be correct!

$$\begin{array}{rcl} 254 \div 16 & = & 15r14 \\ 15 \div 16 & = & 0r15 \end{array}$$

■

If the original decimal number has a fractional portion, the whole number and fractional portion must be converted separately using a slightly different algorithm. First, convert the whole number portion as already described. Take the fractional portion and multiply it by the base. Separate the resulting whole number part and fractional part. The whole number parts are the digits, left to right, which follow the decimal. The multiplication proceeds on each fractional result until the result is entirely a whole number, or you choose to stop (say, in the case of repeating fractional parts).



Different bases differ on whether a particular value is repeating or terminating right of the decimal. This concept will be explored more in the exercises.

■ **Example 9.8** • For example, convert 4.625_{10} into binary (base 2). The whole number portion, 4_{10} , is converted into 100_2 . We then take the fractional portion $0.625 * 2 = 1.25$. The whole number portion is 1, and we repeat the multiplication with the fractional portion,



yielding $0.25 * 2 = 0.5$. The whole number portion is 0, and we repeat the multiplication with the fractional portion, yielding $0.5 * 2 = 1.0$. There is no more fractional portion, so we stop.

$$\begin{array}{rcl} 0.625 * 2 & = & 1.25 \\ 0.25 * 2 & = & 0.50 \\ 0.5 * 2 & = & 1.00 \end{array}$$

The whole number results, from top to bottom, are 101. This becomes the portion right of the decimal. Therefore, $4.625_{10} = 100.101_2$. ■

9.5 Unary

The unary number system (base 1) has only one symbol, and so behaves differently from other bases. It has no representation of zero (except just blank), and the number of marks (instances of the symbol) indicate the value of the number being represented. Unary numbers, or concepts based on them, are occasionally used in theoretical computer science, but not usually in any practical application.



9.6 Exercises

Solutions to these exercises can be found in Appendix A.9 on page 297.

1. *Problem:* An IPv6 address is comprised of eight blocks of four hexadecimal digits. How many bits long is an IPv6 address?
2. *Problem:* Convert the octal number 737_8 into hexadecimal.
3. *Problem:* Convert the binary number 101110_2 into hexadecimal.
4. *Problem:* Convert the hexadecimal number $5D6B_{16}$ into decimal.
5. *Problem:* Convert the number 4033_5 into decimal.
6. *Problem:* Convert the number 51.15_8 into decimal.
7. *Problem:* Convert the decimal number 53.3_{10} into binary.
8. *Problem:* Order the numbers from least to greatest: 123_8 , 123_4 , 123_{16} , 123_{10} .



Chapter 10

Integer Numbers



Integer: positive or negative whole number.

Computers represent all data internally in binary: just binary. There are no built-in representations for a decimal point, or a negative sign. Computers are also expected to perform input and output, which involves converting binary numbers to and from their component decimal digits. The most fundamental computations are those involving integer numbers: positive and negative whole numbers.

Integers are represented internally with a fixed bit width, regardless of the numbers value. Several common sizes are available to the programmer.

Bits	Name
4	nibble
8	byte
16	short
32	long
64	long long

Longer integers may also be supported by a particular programming system, but usually not directly by the processor.

10.1 Unsigned Integers

The simplest type of integer is the unsigned integer. The numeric value of these numbers follows the usual binary conversion with no special cases. The important thing to remember is that there will be a specific width which must be maintained regardless of the numeric value. For example, to represent 12_{10} as an 8-bit unsigned integer, the result would be $0000\ 1100_2$.

The lowest value representable using unsigned integers is zero. The maximum value of an unsigned integer of n bits is $2^n - 1$. The “minus 1” is due to the zero taking up a spot.

 Breaking binary numbers into 4-bit blocks is common for visual purposes. However, there is no actual separation or grouping within the machine.

$\sum_{m=1}^{\infty} \frac{1}{n^m}$ Unsigned integers of n bits have the range $[0, 2^n - 1]$.

10.2 Unsigned Addition

Addition of integer numbers follows the same principles as decimal addition. However, due to base 2, there are only four possible situations per bit pair:

$$\begin{array}{rcl} 0 & + & 0 = 0 \\ 1 & + & 0 = 1 \\ 0 & + & 1 = 1 \\ 1 & + & 1 = 10 \end{array}$$

As in decimal addition, if a two bit result is created, perform a carry. Considering all possible carry situations, the total number of possibilities rises to eight.

$$\begin{array}{rcl} 0 & + & 0 & + & 0 = 0 \\ 0 & + & 1 & + & 0 = 1 \\ 0 & + & 0 & + & 1 = 1 \\ 0 & + & 1 & + & 1 = 10 \\ 1 & + & 0 & + & 0 = 1 \\ 1 & + & 1 & + & 0 = 10 \\ 1 & + & 0 & + & 1 = 10 \\ 1 & + & 1 & + & 1 = 11 \end{array}$$



■ Example 10.1 • For example, to add the 4-bit numbers 0101_2 and 0011_2 , use the normal addition technique and carry whenever a two-bit result is created:

$$\begin{array}{r} & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 \end{array}$$

Thus, $0101_2 + 0011_2 = 1000_2$. This result can be confirmed by converting the binary numbers back to decimal: $5_{10} + 3_{10} = 8_{10}$. ■

■ Example 10.2 • Consider, however, adding the 4-bit numbers 1010_2 and 0111_2 .

 **Overflow:** condition caused when the result of an arithmetic operation is too large for the number of bits available.

 Many programming languages do not indicate an error condition on overflow. The programmer must be careful to avoid situations where overflow may occur, or check for it if the possibility exists.

 Arithmetic overflows have occasionally caused serious problems, such as the crash of Ariane 5 Flight 501. See http://en.wikipedia.org/wiki/Ariane_5_Flight_501

$$\begin{array}{r} & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ \hline 10 & 0 & 0 & 1 \end{array}$$

What to do with the “10” on the left? If there were more bits, we would simply carry the 1. What happens in this case? If there are a fixed number of bits, any bits beyond that limit are lost, and an overflow occurs, indicating that calculation could not be completed successfully. Thus, in 4 bits, $1010_2 + 0111_2 = 0001_2$ with overflow. A processor cannot simply add more bits, as it is physically constructed and wired to use a particular bit size. ■

10.3 Signed Integers

Several techniques for representing negative numbers have been considered. The simplest is to take the left-most bit and make it a sign bit (0 for positive, 1 for negative). Another technique is to invert (replace all 1s with 0s, and all 0s with 1s) all the bits when a negative number is to be represented (known as one's complement). However, both of these techniques have two weaknesses:



1. They both have two distinct values for zero. Specifically, using the sign bit: 0000 and 1000, in 4 bit. Using inversion: 0000 and 1111, in 4 bit. This wastes a sequence and causes one less representable number. It also means special code must be inserted to normalize the zero, since 0 is always 0.
2. They require special handling for addition and subtraction. The signs of each number must be determined and the outcome sign calculated.

In the early days of computing more than now, circuit space was at a premium. Rather than implement the extra circuitry needed for either of these techniques, designers discovered a technique which had neither weakness. This technique has only one representation of zero, and can use the same addition circuits as are used for unsigned numbers for both addition and subtraction of positive and negative numbers.

The Two's Complement system is based on two simple rules:

1. If the leftmost bit is zero, the number is positive.
If the leftmost bit is one, the number is negative.
2. The unsigned (positive) value of a negative number can be found by inverting (replace all 1s with 0s, and all 0s with 1s) all bits and then adding one. This procedure also works identically to find the negative value of a positive number.

Due to the significance of the leftmost bit, it is vital when using Two's Complement to note the bit width and stick to it. For these examples we will use 8-bit Two's Complement. We will use the subscript "2C" to indicate signed Two's Complement numbers.

Numbers with the leftmost bit of zero are treated in the usual way. For example, $0101\ 0101_{2C} = 85_{10}$. Likewise, zero remains $0000\ 0000_{2C} = 0_{10}$.

If the leftmost bit is 1, then the number is negative.

 *Two's Complement:* most common system for representing signed integers in binary; conversion between positive and negative is achieved by inverting and adding one.



Invert: in a binary sequence, replace all 1s with 0s, and all 0s with 1s.



■ Example 10.3 • To find the unsigned value, invert all bits and add one. For example, $1010\ 1010_{2C}$ is negative, so we invert, giving $0101\ 0110$ and add one.

When converting from Two's Complement, remember that the “invert and add one” step is ONLY performed if the number is negative, that is, starts with a 1.



$$\begin{array}{r}
 & & & & & & ^1 \\
 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 \hline
 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0
 \end{array}$$

Thus, $1010\ 1010_{2C} = -0101\ 0110_2 = -86_{10}$. ■

The same technique works in reverse.

■ Example 10.4 • For example, if we want to find the 8-bit Two's Complement representation of -53_{10} , start with the unsigned representation of 53: $-53_{10} = -0011\ 0101_2$. The number is negative, so invert and add one.

Likewise, when converting to Two's Complement, remember that the “invert and add one” step is ONLY performed if the original number is negative.



$$\begin{array}{r}
 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 \hline
 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1
 \end{array}$$

The result is the Two's Complement representation: $-53_{10} = -0011\ 0101_2 = 1100\ 1011_{2C}$. ■

Computers are interested in the unsigned value in binary, so they apply the invert and add one technique. If instead, a direct conversion to decimal is desired, a shortcut exists. Simply put the Two's Complement (positive or negative) into the conversion table and make the leftmost conversion factor negative.

■ Example 10.5 • For example, we can convert the just-found result back into decimal directly.

$$\begin{array}{r}
 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
 -2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0
 \end{array}$$

Add each value.



$$\begin{array}{rcl}
 1 * -2^7 & = & -128 \\
 1 * 2^6 & = & 64 \\
 0 * 2^5 & = & 0 \\
 0 * 2^4 & = & 0 \\
 1 * 2^3 & = & 8 \\
 0 * 2^2 & = & 0 \\
 1 * 2^1 & = & 2 \\
 1 * 2^0 & = & 1 \\
 \hline
 & & -53
 \end{array}$$

The sum is -53, exactly as expected. ■

Two's Complement represents the same number of values as an unsigned field of the same width, but the representation is shifted to allow approximately half negative and half positive values. There is one additional negative number available, because zero takes a spot on the positive side.

$\sum_{n=1}^{\infty} \frac{1}{n^2}$ Two's Complement
 (signed) integers of
 n bits have the range
 $[-2^{n-1}, 2^{n-1} - 1]$.

10.4 Signed Addition and Subtraction

Using Two's Complement, subtraction need not be directly implemented in circuitry. Instead, any computation of the form $a - b$ can be computed as $a + (-b)$, that is, invert and add one to b , and then add to a . Addition operates as specified previously, except that an overflow condition no longer indicates an error in Two's Complement. Instead, an arithmetic error must be detected by checking the outcome of the sign bit and seeing if it meets expectation. Adding two positive numbers should yield a positive, adding two negative numbers should yield a negative result. Adding a positive and a negative cannot result in overflow.

■ **Example 10.6** • For example, consider the problem $22_{10} - 33_{10}$. Subtraction is not supported, so this becomes $22_{10} + (-33_{10})$. Next, convert each value into binary. Two's Complement requires a specific bit width to function correctly; we'll choose eight bits. The problem becomes $0001\ 0110_2 + (-0010\ 0001_2)$. We can then translate these values into Two's Complement. The



first value is positive, so nothing changes. The second value is negative, so invert and add one. This gives the new situation $0001\ 0110_{2C} + 1101\ 1111_{2C}$.

$$\begin{array}{r}
 & & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
 \hline
 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1
 \end{array}$$

$\sum_{n=1}^{\infty} \frac{1}{n^2}$

With any number of bits, if a Two's Complement number is all 1s, then the value is -1 .

The result of the addition is $0001\ 0110_{2C} + 1101\ 1111_{2C} = 1111\ 0101_{2C}$. The leftmost bit is one, so the entire number is negative. To find the unsigned value, we can invert and add one. Thus, $1111\ 0101_{2C} = -0000\ 1011_2 = -11_{10}$. ■

■ **Example 10.7** • To see how the overflow signal no longer indicates an arithmetic error, consider $-1_{10} + -1_{10}$. For simplicity, we'll do this one in four bits. The number is negative, so invert (0001_2 becomes 1110) and add one, giving 1111_{2C} . We now have $1111_{2C} + 1111_{2C}$.

$$\begin{array}{r}
 & & 1 & 1 & 1 \\
 1 & 1 & 1 & 1 & 1 \\
 1 & 1 & 1 & 1 & 1 \\
 \hline
 11 & 1 & 1 & 0
 \end{array}$$

The extra 1 is thrown away, and triggers an overflow condition. However, the overflow condition is not significant here, as 1110_{2C} is the correct result. Instead of checking the overflow condition, the sign bit consistency should be checked instead whenever dealing with Two's Complement. ■

■ **Example 10.8** • An example of a Two's Complement arithmetic error can be demonstrated by attempting to perform $7_{10} + 6_{10}$ in four-bit Two's Complement. Each of these numbers is positive, and so convert directly into $0111_{2C} + 0110_{2C}$

$$\begin{array}{r}
 & & 1 & 1 \\
 0 & 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 0 \\
 \hline
 1 & 1 & 0 & 1
 \end{array}$$



Note that no overflow occurs in this situation. However, the result of 1101_{2C} is negative (because the leftmost bit is 1), which seems unlikely given the addition of two positive numbers. This is how checking the sign bit can detect Two's Complement arithmetic errors. ■

10.5 Binary Multiplication

Binary multiplication is performed by repeated shifting and addition. Shifting refers to the process of adding or removing the rightmost bit of a binary number. For multiplication, a left shift (adding a zero to the right side of a binary number) is used. For each 1 bit in the number to be multiplied, shift that number by the bit's position (with the rightmost bit being a shift of zero), and sum all the results. Left shift is often denoted $x \ll p$ where x is the number to be shifted and p is how many positions to shift it. For example, $1101_2 \ll 10_2 = 110100_2$. Left and right shifts, in computing, are always performed at the binary level, regardless of the bases of the inputs. So $13_{10} \ll 2_{10} = 52_{10}$.

■ **Example 10.9** • For example, multiply $3_{10} * 5_{10}$ in binary. Start with $3_{10} = 11_2$ and $5_{10} = 101_2$. Then find $11_2 * 101_2$. In this case, note that the number 101_2 has a 1 bit in the first place and the third place. Thus, shift 11_2 by zero and two, respectively, obtaining $11_2 \ll 2_{10} = 1100_2$ and $11_2 \ll 0_{10} = 11_2$. Add these results together to get the final answer: $1111_2 = 15_{10}$. ■

This technique works correctly for Two's Complement numbers as well, as long as there is enough room to store the result. When left shifting a fixed width number, discard bits that fall off the left as zeros are added to the right to maintain the same width.

■ **Example 10.10** • For example, multiply $-4_{10} * 3_{10}$ in eight bit Two's Complement. Start with $-4_{10} = 1111\ 1100_{2C}$ and $3_{10} = 0000\ 0011_{2C}$. The number $0000\ 0011_{2C}$ has a 1 bit in the first and the second place. Thus, shift $1111\ 1100_{2C}$ by zero and one, respectively, and add the results: $1111\ 1100_{2C} + 1111\ 1000_{2C}$

 Left shifting a fixed width value causes the bit(s) on the left to simply “fall away”. Do not extend the width of the number, as the computer does NOT gain any additional width during these computations.

$\sum_{n=1}^{\infty} \frac{1}{n^8}$

Multiplying by a power of 2 can be accomplished in a single step by shifting left by that power. In other words $x * 2^n = x \ll n$

$$\begin{array}{cccccccc}
 & 1 & 1 & 1 & 1 \\
 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
 \hline
 11 & 1 & 1 & 1 & 0 & 1 & 0 & 0
 \end{array}$$

The extra 1 bit falls off, and the result is $11110100_{2C} = -12_{10}$.

Multiplication is commutative, so this example could be done as $3_{10} * -4_{10}$ instead. In that case, we would use $1111\ 1100_{2C}$ to determine how much to shift. The number has a 1 bit in the third through eighth places, indicating shifts of two through seven. The number to be shifted is $0000\ 0011_{2C}$.

$$\begin{array}{cccccccc}
 & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 11 & 1 & 1 & 1 & 0 & 1 & 0 & 0
 \end{array}$$

Although considerably more shifting and addition was needed for this process, the result is the same: $1111\ 0100_{2C} = -12_{10}$. ■

10.6 Binary Coded Decimal

$\sum_{n=1}^{\infty} \frac{1}{n^8}$

The number of bits required for a BCD representation of a base-10 number is $\lceil \log_{10}(n+1) \rceil * 4$ (if n is in base 10).

Numbers are input and output in decimal, one digit at a time. Binary coded decimal (BCD) is a technique for representing decimal numbers in binary based on their digits. Each digit is represented with a 4-bit block, using the normal binary conversions (0000 for 0 up through 1001 for 9). For example, $53_{10} = 0101\ 0011_{BCD}$. Some conventions of BCD take advantage of the remaining blocks (1010 through 1111) to define control sequences, or to indicate positive/negative numbers.

Conversion from BCD to binary involves looking up a binary multiplier for each position (the digit value) and



performing binary multiplication. Take each 4-bit digit block, and multiply it by the position value (rightmost digit is 1_2 , next digit is 1010_2 , next digit is 1100100_2 , and so on) and sum the results.

■ **Example 10.11** • For example, to convert $0101\ 0011_{BCD}$ into binary, first multiply each 4-bit block by the position value. This gives $0101_2 * 1010_2$ and $0011_2 * 1_2$. Any value multiplied by 1 is just itself, so the only “real” multiplication to solve is $0101_2 * 1010_2$. The number 1010_2 has a 1 bit in the second and fourth place, indicating a shift of one and three, respectively.

$$\begin{array}{r} & 1 \\ & 1 \ 0 \ 1 \ 0 \\ \times & 1 \ 0 \ 1 \ 0 \ 0 \ 0 \\ \hline 1 \ 1 \ 0 \ 0 \ 1 \ 0 \end{array}$$

The sum $110010_2 = 50_{10}$ gives that particular digit. We then add the other digit, 0011_2

$$\begin{array}{r} & 1 \\ & 1 \ 0 \ 0 \ 1 \ 0 \\ & 1 \ 1 \\ \hline 1 \ 1 \ 0 \ 1 \ 0 \ 1 \end{array}$$

Thus, the value $0101\ 0011_{BCD} = 110101_2 = 53_{10}$.

Conversion from binary to BCD is accomplished using the double dabble algorithm, which involves shifting and addition. The double dabble algorithm is fast and straightforward, but does require some mental gymnastics to consider bits as varying representations throughout the transformation.

To set up for the double dabble algorithm, a numeric space is prepared by determining how many decimal digits will be required. Each decimal digit is represented initially by four zero bits. These bits are followed by the original binary representation.

The algorithm proceeds iteratively for as many bits as the original number has. At each step:

$\sum_{n=1}^{\infty} \frac{1}{n^2}$

The number of bits required for a BCD representation of a base-2 number of n bits is $\lceil \log_{10} 2^n \rceil * 4$.



Double Dabble: a shift and add algorithm to translate binary into BCD.



Rather than pre-calculating the number of BCD blocks required, it is possible to dynamically allocate them by adding a block whenever a left shift would push 1 off the edge. Alternatively, and as occurs in most practical applications, a preset fixed number of blocks are allocated. For example, on a digital clock, the minute part always has two blocks.

1. For any BCD block greater than 4, add 3 to it.
(Note the result may not be valid BCD, but this is not a problem)
2. Shift left the entire space.

■ **Example 10.12** • For example, to convert the binary number 11010011_2 into BCD, we would first determine that three decimal digits are needed. The initial space is thus 0000 0000 0000 11010011. The three blocks will become the three BCD digits, but the entire space will be used for shifting.

In order to track when the bits are consumed, we will not add zeros on the right hand side when shifting left. This way, when all of the initial number is gone, it will be clear because only the BCD blocks will remain.

	BCD Blocks			Input	Operation
1.	0000	0000	0000	11010011	Start
2.	0	0	0		none > 4
3.	0000	0000	0001	1010011	Shifted Left
4.	0	0	1		none > 4
5.	0000	0000	0011	010011	Shifted Left
6.	0	0	3		none > 4
7.	0000	0000	0110	10011	Shifted Left
8.	0	0	6		Rightmost block exceeds four
9.	0000	0000	1001	10011	Added 11_2 to rightmost block
10.	0000	0001	0011	0011	Shifted Left
11.	0	1	3		none > 4
12.	0000	0010	0110	011	Shifted Left
13.	0	2	6		Rightmost block exceeds four
14.	0000	0010	1001	011	Added 11_2 to rightmost block
15.	0000	0101	0010	11	Shifted Left
16.	0	5	2		Middle block exceeds four
17.	0000	1000	0010	11	Added 11_2 to middle block
18.	0001	0000	0101	1	Shifted Left
19.	1	0	5		Rightmost block exceeds four
20.	0001	0000	1000	1	Added 11_2 to rightmost block
21.	0010	0001	0001		Shifted Left, final result

Thus, $11010011_2 = 0010\ 0001\ 0001_{BCD} = 211_{10}$. ■



10.7 Exercises

Solutions to these exercises can be found in Appendix A.10 on page 303.

1. *Problem:* Show how the computer performs the unsigned addition $45_{10} + 17_{10}$ in eight bit binary.
2. *Problem:* What happens when the computer attempts to perform the unsigned addition $201_{10} + 99_{10}$ in unsigned eight bit binary? How can the error be detected?
3. *Problem:* Show how the computer represents -77_{10} in eight bit Two's Complement.
4. *Problem:* Show how the computer performs the subtraction $13_{10} - 23_{10}$ in eight bit Two's Complement.
5. *Problem:* Show how the computer performs the addition $-20_{10} + 23_{10}$ in eight bit Two's Complement.
6. *Problem:* What happens when the computer attempts to perform the addition $-100_{10} + (-80_{10})$ in eight bit Two's Complement? How can the error be detected?
7. *Problem:* Convert the number 354_{10} into binary via BCD.
8. *Problem:* Convert the number 287_{10} into BCD via binary.

Chapter 11

Floating Point Numbers

 *Floating Point:* representation of a number using a form of scientific notation, where the position of the decimal point is separated from the numeric digits.

Early computers could only calculate based on integers. Calculations with fractional values are important to many fields which used computers, such as science, engineering, and economics, so development of a technique for representing and computing with fractional values occupied many developers' minds. The main requirements for fractional values were a large representational range (as users might be using very small values, and others might be using very large values), and the ability to implement the mathematical operations into hardware. Today, two primary floating point representations exist (binary and decimal), both codified in the IEEE-754 standard.

11.1 Scientific Notation

Floating point numbers apply the concept of scientific notation. Any number large or small can be represented in scientific notation using an exponent. For example, $35909 = 3.5909 * 10^4$. Likewise, $0.00000543 = 5.43 * 10^{-6}$. This form, with exactly one non-zero digit left of the decimal, is known as normalized exponential form.

The general form of any scientific notation is $\pm s * b^e$, where s represents the significant digits, b is the base,

and e is the exponent. When computers output in base 10 and exponents are involved, the letter e is sometimes used. For example, $5.43 * 10^{-6}$ may be output as $5.43e-6$.

Like integer numbers, with a fixed number of bits available, there are limits to the range of numbers that can be represented. In floating point numbers, these limits appear both in the exponent and the significant digits. Limitations of the exponent restrict the upper and lower range of the number, while limitations of the significant digits restrict how much rounding or approximation is required to represent a number.

Binary floating point numbers are based on binary scientific notation. Numbers are generally converted from standard form decimal into binary, and then into scientific notation.

■ **Example 11.1** • For example, consider the value -200.625_{10} . Recall the procedure for converting a decimal value into binary involves converting the whole number and fractional portion separately. Multiply the fractional portion repeatedly by 2, separating and retaining the whole number 1 or 0 as the digits.

In this case, we take the fractional portion $0.625 * 2 = 1.25$. The whole number portion is 1, and we repeat the multiplication with the fractional portion, yielding $0.25 * 2 = 0.5$. The whole number portion is 0, and we repeat the multiplication with the fractional portion, yielding $0.5 * 2 = 1.0$. There is no more fractional portion, so we stop. The whole number results, from left to right, are 101.

Combined with the left hand side, where $200_{10} = 11001000_2$, we have an entire conversion of $-200.625_{10} = -11001000.101_2$. To represent this value in scientific notation, we shift the decimal point until a single digit (bit) remains left of the decimal place. The count of how many shifts are needed forms the exponent. In this case, we must move the decimal place left 7 places.

The final representation is $-11001000.101_2 = -1.1001000101_2 * 2^7$. Note that the base is 2 in-



The conversion of fractional decimals into binary is discussed in the chapter on number systems.

Given a number in normalized exponential notation, in the form $\pm s * b^e$, if $e < 0$ then the entire value is in the range $(-1, 1)$. If $e \geq 0$, then the value is outside that range.

$$\sum_{n=1}^{\infty} \frac{1}{n^3}$$



Significand: portion of a floating point number consisting of the significant digits.



Bias: the amount a stored value is offset from its actual value.



Implicit Bit: the leftmost one in binary normalized exponential form that is assumed to be present but not actually stored.

stead of 10, as the significant digits are in binary instead of decimal. ■

■ **Example 11.2** • Likewise, consider the representation of 0.09375_{10} into binary scientific notation. Converting the fractional portion into binary gives 0.00011_2 . To convert to scientific notation, the decimal point must move right four spaces, yielding $1.1 * 2^{-4}$. ■

11.2 Binary Number Composition

Floating point numbers are always a fixed bit width, and are composed of three parts: a sign bit, an exponent, and the significant digits. The significant digits portion of the number is often called the mantissa, significand, or coefficient. For each given bit width, the distribution of bits between the exponent and significand are fixed; the sign bit is always a single bit. The bulk of the bits are usually given to the significand. In addition, the exponent (which may be positive or negative) is not represented using Two's Complement, but using a separate technique known as biasing.

Exponent bias works by adding the bias to the exponent, and then storing the result as an unsigned integer. When the number is processed, the bias is subtracted from the exponent value to find the actual exponent. This allows positive and negative exponents to be converted into unsigned integers. The bias is always added, regardless of whether the original exponent is positive or negative.

In all binary numbers except 0, normalized exponential form will place a 1 to the left of the decimal place. As this 1 will always be present (except in the case of 0), it is not stored in the floating point number, saving one bit. Thus, the number of significand bits represented is one more than actually stored.

With these rules, the procedure for representing a number in binary floating point is the same for all bit widths, except that the appropriate sizes and exponent

bias must be used.

The IEEE standard defines four binary floating point sizes: 16-bit (half precision), 32-bit (single precision), 64-bit (double precision), and 128-bit (quadruple precision). Double precision floating point is the most common form.

Total Bits	Exponent Bias	Exponent Bits	Sig. Bits
16	15	5	10
32	127	8	23
64	1023	11	52
128	16383	15	112

The sign bit (1 for negative, 0 for positive) is always the leftmost (most significant) bit. The exponent follows, and the significand comes last. The rightmost (least significant) bit of a floating point number corresponds to the rightmost (least significant) bit of the significand.

■ **Example 11.3** • Consider representing the simple number $1_{10} = 1_2 = 1.0_2 * 2^0$ in 16-bit floating point. The number is positive (sign bit = 0), the biased exponent is $0_{10} + 15_{10} = 15_{10} = 1111_2$, and the significand is all zeros. Recall that the one that is left of the decimal place is not represented in the significand, it is an implicit bit.

Be sure that each component exactly fills (padding or truncating as needed) its allotted bits. The biased exponent is allotted five bits, so it will be 01111. The significand is allotted 10 bits, so it will be 00 0000 0000. Thus, the final representation is 0011 1100 0000 0000. The five underlined bits represent the biased exponents: the bit to the left is the sign bit, and the bits to the right are the significand. Due to the size of floating point numbers, they are often shown in hexadecimal, in this case, $3C00_{16}$. ■

■ **Example 11.4** • Taking another example started earlier, we found that $-200.625_{10} = -1.1001000101_2 * 2^7$. This number can also be represented in 16-bit floating point. The number is negative (sign bit = 1), the biased exponent is $7_{10} + 15_{10} = 22_{10} = 10110_2$, the significand

$\sum_{n=1}^{\infty} \frac{1}{n^s}$
If n is the number of exponent bits, then the exponent bias is $2^{n-1} - 1$.

$\sum_{n=1}^{\infty} \frac{1}{n^s}$
The total number of bits in a floating point number is the exponent bits plus the significand bits, plus one for the sign bit.



There is no innate distinction within the computer as to which bits form the sign, exponent, or significand. Instead, the processor knows what kind of number it is expecting, and it knows how many bits are used for each part, and so it divides up the number into the appropriate parts.



If the binary representation of the biased exponent exceeds the number of exponent bits allotted, or is the biased exponent is negative, the number is too large or too small to be represented with the number of bits available.

is up to ten bits right of the decimal point. In this case, there are exactly ten bits right of the decimal point, so the significand is 1001000101.

Attaching these three pieces together, the final floating point representation is 1101 1010 0100 0101, or in hex as $DA45_{16}$. Again, the underlining delineates the exponent portion from the sign and significand. The underline itself appears for clarity only and is not actually stored with the number. ■

Converting from floating point representation back into an actual number is simply the reverse process. Remember to subtract, rather than add, the exponent bias on the conversion back.

■ **Example 11.5** • For example, to convert the 16-bit floating point number $ABCD_{16}$ into its value, start with the binary equivalent: 1010 1011 1100 1101.

Identify the portions, including the sign bit, biased exponent, and significand: 1010 1011 1100 1101. The sign is 1, so the number is negative. The exponent is $1010_2 = 10_{10} - 15_{10} = -5_{10}$. Remember to place the significand right of the decimal place, and the implicit 1 left of the decimal place. This gives the scientific notation $-1.1111001101_2 * 2^{-5}$.

If desired, this value could be converted into standard notation: $-1.1111001101_2 * 2^{-5} = -0.00001111001101_2$ and even back into decimal.

0	0	0	0	1	1	1	1	1	0	0	1	1	0	1
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

Add up each place.



$$\begin{array}{rcl}
 0 * 2^{-1} & = & 0 \\
 0 * 2^{-2} & = & 0 \\
 0 * 2^{-3} & = & 0 \\
 0 * 2^{-4} & = & 0 \\
 1 * 2^{-5} & = & 0.03125 \\
 1 * 2^{-6} & = & 0.015625 \\
 1 * 2^{-7} & = & 0.0078125 \\
 1 * 2^{-8} & = & 0.00390625 \\
 1 * 2^{-9} & = & 0.001953125 \\
 0 * 2^{-10} & = & 0 \\
 0 * 2^{-11} & = & 0 \\
 1 * 2^{-12} & = & 0.000244140625 \\
 1 * 2^{-13} & = & 0.0001220703125 \\
 0 * 2^{-14} & = & 0 \\
 1 * 2^{-15} & = & 0.000030517578125 \\
 \\
 & & \hline
 & & 0.060943603515625
 \end{array}$$

Attach the whole number portion (none in this case), and apply the sign bit. The final value is -0.060943603515625_{10} . Of course, if the application is satisfiable with a smaller number of significant digits, say three or four, the process could be somewhat abbreviated. ■

11.3 Special Cases

There are a few cases which do not follow these rules: these cases are denoted by an exponent block of all 0s or all 1s. If the exponent block is 0, this indicates either the value of zero (if the significand is zero) or a subnormal number. If the exponent block is all 1s, this indicates an error condition: either an infinity value (if the significand is zero), or “not a number” (NaN). Infinities and NaNs are generated by operations which exceed the representational capability of the number, or which do not have a defined result (such as dividing by zero).

Unlike normalized numbers, subnormal numbers do not have the implicit 1 bit left of the decimal point. Instead, a 0 bit is used. This allows numbers smaller than would normally be allowed by the exponent limit. For



Subnormal Number: non-normalized numbers that fill in near zero to help avoid truncating to zero.

example, in 16-bit floating point, the smallest positive normalized number would be 0000 0100 0000 0000. The biased exponent is 1_{10} , subtract the exponent bias to find the actual exponent: $1_{10} - 15_{10} = -14_{10}$. Place the implicit 1 left of the decimal point, and you have the smallest normalized positive value: $1.0 * 2^{-14}$.

If we switch to subnormal numbers, then the implicit 1 left of the decimal goes away. The smallest positive subnormal number would be 0000 0000 0000 0001. The exponent is converted in the usual way: $0_{10} - 15_{10} = -15_{10}$, however, NO implicit 1 is placed left of the decimal, giving: $0.0000 0000 01 * 2^{-15} = 1.0 * 2^{-24}$.

Every binary floating point value can be considered to be a fraction in the form $\frac{a}{2^b}$. If the value to be represented cannot be represented in decimal with a denominator of 2^b , then no exact floating point representation is possible, no matter how many bits are allotted.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

11.4 Floating Point Errors

Although some numbers can be exactly represented in floating point, most numbers require rounding. Even innocuous looking numbers like 3.3_{10} cannot be exactly represented in binary floating point, the conversion from 0.3_{10} into binary is not exact. The floating nature of the decimal point also means that certain operations will produce different results if they are performed in a different order. Consider some actual outputs using 32-bit floating point (all numbers shown are base 10):

- $1.1 + 2.2 = 3.3000002$
- $1.1 + 2.2 - 2.2 = 1.1000001$
- $1.2 - 1.1 = 0.100000024$
- $1.2 - 1.1 == 0.1 = \text{false}$
- $1 + 1000000000 = 1000000000$

These errors seem very small, but they compound when many operations are used. Consider adding together ten million numbers: all the numbers are 0.1. This should be the same as $0.1 * 10\,000\,000 = 1\,000\,000$. Consider what actually happens when this operation is performed as ten million additions rather than a multiplication. Throwing a larger number into the mix also has unexpected results:



- With ten million 0.1 values, $0.1 + 0.1 + 0.1 + \dots + 0.1 = 1087937.0$
- Beginning with a large number, and retaining the ten million small numbers, $1000000 + 0.1 + 0.1 \dots + 0.1 = 2097152.0$
- Ending with a large number instead, $0.1 + 0.1 + 0.1 \dots + 0.1 + 1000000 = 2087937.0$

These errors are not programming language specific, and do not go away with additional bits (although additional bits decrease the amplitude of the errors). Therefore, whenever binary floating point is in use, a few guidelines should always be kept in mind:

1. *Always treat a floating point number as an approximation.* Although some numbers can be exactly represented, you should never assume that the number you have is exact.
2. *Never compare floating point numbers for equality.* The “same” mathematical operations can produce slightly different floating point values, depending on the operation ordering. The system may be rounding values so that they look the same, but may be different internally. Instead, check the absolute difference to see if it is within some acceptable range.
3. *When summing a list, always start with the smallest values.* Summing smallest to largest will minimize the total error accumulated.
4. *Floating point numbers are unsuitable for financial calculations* or any other field where numbers must be exact.



Scientific measurements, which are often of limited precision to begin with, are a good candidate for binary floating point numbers.

11.5 Decimal Floating Point

In order to avoid the problems associated with binary floating point numbers, increasing computing power



has allowed the introduction of decimal floating point numbers. These numbers follow generally the same structure as binary floating point numbers (sign bit, exponent bits, significand), but instead of representing the significand with standard binary, a form of binary coded decimal is used instead.

In order to be able to represent a similar range of values, the size of the exponent and significand are somewhat flexible, and a smaller exponent allows for a few more significand bits, which is needed to represent certain decimal digits. Using regular BCD would be very wasteful: each BCD digit requires 4 bits. Note $2^4 = 16$, but only 10 unique digits exist. Thus, regular BCD utilizes only 62.5% of the bit space. To overcome this weakness, decimal floating point numbers are represented using densely packed decimal.

Densely packed decimal is a form of BCD where 10 bits represent three decimal digits. A quick calculation shows that $2^{10} = 1024$, and three decimal digits have 1000 possibilities (000 thru 999), yielding a 97.7% space utilization.

The advantage of decimal floating point is that any number which can be exactly represented in decimal with the specified number of digits (for example, 0.1) can be exactly stored. The disadvantage of decimal floating point is that significant additional computation is required to unpack and repack the densely packed decimal and perform the operations. Thus, decimal floating point can be expected to perform slower than binary floating point.



The programming language used to perform this test did not natively support 128-bit binary floating point, or 64-bit or less decimal floating point.

On a current computer system, summing up the number 0.1 a million times was found to take about ten times longer using the decimal data type as compared to a binary floating point number. However, the sum that was calculated using the binary floating point numbers encountered a cumulative error, whereas the decimal number did not.

Data Type	Time Taken (ms)	Sum Result
32-bit binary	43	1087937
64-bit binary	39	999999.999838975
128-bit decimal	412	1000000.0

The performance limitations of decimal floating point are somewhat lessened by the inclusion of decimal floating point operations directly into most modern processors. Increasing processor speed has also allowed these operations to be performed at a reasonable rate, making decimal floating point the logical choice for most computations.

Decimal floating point numbers help reduce errors substantially, but are still subject to limitations. In particular, any decimal number which requires more digits than the data type can store will incur rounding. The 128-bit decimal data type can store 34 decimal digits.



11.6 Exercises

Solutions to these exercises can be found in Appendix A.11 on page 309.

1. *Problem:* Convert 14.25_{10} into 16-bit binary floating point.
2. *Problem:* Convert -2.67_{10} into 16-bit binary floating point.
3. *Problem:* Convert $\frac{1}{3}$ (base 10) into 16-bit binary floating point.
4. *Problem:* Convert the 16-bit floating point number 9876_{16} into decimal.
5. *Problem:* Convert the 16-bit floating point number $7BFF_{16}$ into decimal.

What, if anything, is special about this value?

6. *Problem:* What is the smallest positive integer which *cannot* be represented in 16-bit binary floating point?
7. *Problem:* In 16-bit binary floating point, how many numbers can be represented between 2_{10} and 3_{10} , inclusively?
8. *Problem:* In 16-bit binary floating point, how many numbers can be represented between 0_{10} and 1_{10} , inclusively?



Chapter 12

Unicode and ASCII

In addition to numbers, computers needs to represent letters and words to provide output and receive input from people. Historically, a variety of representations have been used. Relevant standards for English speaking countries are ASCII and Unicode.

12.1 ASCII

ASCII is a historical representation, derived from telegraph operations before digital computers were in use. It remains interesting today because most modern encodings, such as Unicode, encompass at least a significant portion of ASCII.

ASCII started life as a 7-bit encoding. The first 32 characters (00_{16} thru $1F_{16}$) consist of “non-printing” characters or “control” characters. The control characters included special typewriting operations and commands to devices, such as printers. For example, character $0B_{16}$ represents a tab, and character 08_{16} represents a backspace. For printers, character $0C_{16}$ represents a form feed (new page) while character $0A_{16}$ represents a line feed. These codes made sense in the days of rotary printers, such as dot-matrix, but have little relation to the page-at-a-time printers of the modern day.

The next 32 characters (20_{16} thru $3F_{16}$) consist of digits

 *ASCII:* short for American Standard Code for Information Interchange, a 7 or 8 bit encoding of English letters, digits, punctuation, and other symbols.



When referring to particular ASCII codes, it is common to reference them in two digit hexadecimal, even though a maximum of 7 bits may be involved.

and punctuation. Capital letters begin at 41_{16} , and lowercase letters begin at 61_{16} . The following table shows all characters in 7-bit ASCII. The shaded characters are control characters. The octal digit on the left side identifies the most significant three bits, while the hexadecimal digit on top identifies the less significant four bits. The blank character at 20_{16} is the space, as would appear when the space bar was pressed on the keyboard.

ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0 NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1 DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2 !	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3 @	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4 @	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5 P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
6 `	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7 p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

For example, if the value $100\ 1010_2$ represents a character in ASCII, that character is $001\ 1010_2 = 4A_{16}$ which is the symbol J . If we wanted to represent the symbol f using ASCII, it would be represented as $66_{16} = 111\ 0111_2$. The last character in 7 bit ASCII ($7F_{16}$) is a control character for the delete key.



Extended ASCII: any 8-bit or more extension of ASCII encoding.

Seven bit ASCII was acceptable for simple English messages. However, as computer systems developed, the need to represent other languages, such as Spanish, to display special symbols such as £ , and also the desire to provide visual decoration increased. As a result, an eight bit version of ASCII was developed. The 8-bit encoding shares the first 128 symbols with the original ASCII, and defines an additional 128 more in the range of 80_{16} thru FF_{16} .

Several variations of extended ASCII were proposed, with conflicting characters chosen for the upper 128 range. Users had to be careful to select the proper encoding to indicate which version of extended ASCII was in use, otherwise the extended characters would appear as gibberish. In addition, it became clear that 128 characters was nowhere near enough to represent more pictographically complex languages such as Mandarin or Japanese. The Internet, bringing computing and communication to the global scale, requiring a



single character encoding which could represent all the world's messages. ASCII was not up to the job.

12.2 Unicode

Unicode was created as a way to represent characters without arbitrary limits. For reasons of backwards comparability, Unicode adopted the first 128 ASCII symbols as its first 128 symbols. Unicode is divided into two parts: code points (which are simply the pairing of a symbol with a number), and an encoding (which describes how to store the numbers).

Unicode code points are usually represented as U+0123 where the "U+" indicates that the number is a Unicode code point, and the four digits are four hexadecimal digits indicating the code point. For example, U+004A is the symbol J. This is true because the first 128 Unicode code points match the 128 7-bit ASCII characters. The four hexadecimal digits provide over 65,000 code points; however, this is not a limit. Unicode has currently defined over 100,000 code points; the use of four hexadecimal digits is merely convention. The symbol J could also be represented as U+00004A or U+4A.

The encoding and storage of Unicode code points, however, was not obvious. Assume we wanted to store U+004A (capital J) followed by U+004C (capital L). We could store this as $4A\ 4C_{16}$. How would the reader know that refers to U+004A followed by U+004C, and not the single character U+4A4C?

In order to resolve this issue, a variety of Unicode encodings have been considered. Among the earliest was UCS-2, which used 2 bytes (16 bits) for each character, and supported only up to 65,536 code points. UCS-2 was troublesome for several reasons. A major problem was that much of the data being transmitted and stored was in English, and thus could be stored using only one byte. Developers who used English exclusively complained that UCS-2 wasted space; a UCS-2 string took twice as many bytes as the equivalent ASCII string.



Code Point: a number which represents a specific symbol in Unicode.



Character Encoding: a transformation which describes how to store a particular code point in memory or on disk.



Complete charts of all Unicode symbols are available at <http://www.unicode.org>



UCS-2: an early Unicode encoding which represented each code point with 16 bits.



For example, in eight bit ASCII, the word “Hello” is represented as $48\ 65\ 6C\ 6C\ 6F_{16}$, requiring a total five bytes. In UCS-2, the same “Hello” is represented as $00\ 48\ 00\ 65\ 00\ 6C\ 00\ 6C\ 00\ 6F_{16}$, requiring ten bytes. Although this example seems insignificant, when very large amounts of text were being stored or processed, the overhead was simply too much.

Another problem with UCS-2 became apparently as the Unicode standard reached and exceeded 65,536 code points. UCS-2 had no way to represent the high code points. Like ASCII, UCS-2 was not flexible enough and is considered obsolete.

A variety of other encodings were proposed, and used to varying degrees. Today, however, almost all Unicode documents are encoded using UTF-8, which combines the compactness of ASCII with the flexibility to represent any code point. UTF-8 uses 8-bit bytes to represent data; if 7-bit ASCII is used with a leading bit of 0, UTF-8 is backwards compatible with ASCII. In addition, UTF-8 can represent over one million code points. As a result of this compactness and flexibility, UTF-8 is largely the standard Unicode encoding.



UTF-8: modern Unicode encoding which is backwards compatible with 7 bit ASCII and can represent code points up to U+10FFFF.

The number of bytes required for UTF-8 is one for representing a code point up to 127, and increases up to four for the highest code points. The table below shows how to encode code points up to U+FFFF into UTF-8. The letters in the patterns each match one bit, and show how that bit from the code point would appear in the UTF-8 encoding.

Unicode Code Point	Code Point in Binary	Code Point Pattern	UTF-8 Encoding
U+0000 to U+007F	0000 0000 to 0111 1111	0abc defg	0abc defg
U+0080 to U+07FF	0000 1000 0000 to 0111 1111 1111	0abc defg hijk	110a bcde 10fg hijk
U+0800 to U+FFFF	0000 1000 0000 0000 to 1111 1111 1111 1111	abcd efgh ijkl mnop	1110 abcd 10ef ghij 10kl mnop

Using UTF-8, “Hello” is encoded exactly the same as it would be in ASCII.

■ **Example 12.1 • To encode the Euro sign (U+20AC),**

UTF-8 would use three bytes. The code point $20AC_{16} = 0010\ 0000\ 1010\ 1100_2$. Applying this to the pattern above, the first byte is $1110\ \underline{0010}_2$, followed by $1000\ \underline{0010}_2$. The last byte is $1010\ \underline{1100}_2$. The Euro sign is represented in UTF-8 as $E2\ 82\ AC_{16}$. ■

To read UTF-8 and convert it into Unicode code points, the leftmost bits of the first byte must be considered. If the leftmost bit is 0, then a single byte is consumed and transformed directly into a code point. If the leftmost bits are 110, then two bytes are consumed and transformed according to the second pattern to create a code point in the U+0080 to U+07FF range. If the leftmost bits are 1110, then three bytes are consumed and transformed according to the third pattern to create a code point in the U+0800 to U+FFFF range. A similar pattern exists for the higher code points, using four bytes. The leftmost bit pattern of 10 is used to indicate this byte is part of a larger pattern.

■ **Example 12.2** • For example, to decode the UTF-8 sequence $C2\ A2_{16}$, first transform it into binary: $1100\ 0010\ 1010\ 0010_2$. Starting with the first byte, match the leftmost bits: 110 indicates that two bytes will be used with the second pattern. The two bytes match $1100\ \underline{0010}\ 1010\ \underline{0010}_2$. Taking the underlined parts (which are the code point portions), and prepending the 0s from the pattern, we find the Unicode code point is $0000\ \underline{1010}\ \underline{0010}_2 = 0A2_{16}$. The Unicode code point U+00A2 refers to the “cent” sign. ■

The code point for the Euro sign ends in AC_{16} , as does the UTF-8 encoding. However, the “A” part is coincidence. In general, only the last four bits will always match between a code point and its UTF-8 encoding.



12.3 Display Issues

In order to display any character on the screen, the code point must be replaced with a symbol. The symbol is looked up from a table; the table of symbols is known as a font. Most computers include a variety of fonts, such as serif, sans-serif, with bold and italic versions, along with more fanciful fonts. In the process of creating a font, the artist creates a shape for each symbol that the font can display. When dealing with ASCII, only a small and fixed number of symbols need to be created. However, the number of symbols in Unicode



Font: a set of symbols which represent various code points.



Unicode defines a “replacement character” U+FFFD (the box) which can be used to indicate that a symbol is unavailable for display or invalidly encoded.

is large and growing.

Many fonts support only a very limited subset of Unicode. Some fonts, designed and indicated as Unicode fonts, support a much larger base. However, few if any fonts support all Unicode code points. Thus, even though a particular symbol is listed in the Unicode standard and has a representable code point, it may not display correctly on the screen.

If an application is likely to include a variety of languages, care should be taken not just to support Unicode over ASCII, but to use or make available appropriate fonts which can render the various language symbols involved.

Attempts to render a symbol not supported by a particular font usually appear as \square or $?$.

Another minor issue in display is that numbers, represented internally as unsigned or signed integers, or floating point, must be converted into a sequence of symbols to be displayed. This normally involves a conversion through BCD, which can then be mapped from one BCD block into one display symbol. Input of numbers works similarly, with a reverse process.

12.4 Exercises

Solutions to these exercises can be found in Appendix A.12 on page 314.

1. *Problem:* Show how the phrase “Tall tree” would appear in UTF-8.
2. *Problem:* Find the UTF-8 encoding for the Unicode code point 2399_{10} .
3. *Problem:* Find the Unicode code point associated with the UTF-8 encoding $D7\ 91_{16}$.
4. *Problem:* A certain document contains 1,500 English letters, 350 standard punctuation marks (including space), and 50 symbols in Arabic. Arabic symbols have Unicode code points from U+0600 through U+06FF. Determine how many bytes of disk space this document will take to store using UCS-2 compared to UTF-8. (Assume no overhead for formatting; only the encoded characters will be stored).
5. *Problem:* The “Heart Sutra” is a famous Buddhist text. It is very short, at only 260 Chinese symbols. Chinese symbols have Unicode code points from U+4E00 through U+9FCF. Determine how many bytes of disk space this document will take to store using UCS-2 compared to UTF-8. (Assume no overhead for formatting; only the encoded characters will be stored).
6. *Problem:* One English translation of the “Heart Sutra” contains 1,357 English symbols. Will this document (again, assuming no overhead), encoded in UTF-8, take more or less space than the Chinese version, also encoded in UTF-8?
7. *Problem:* The word “hello” in Chinese is rendered as 你好. In UTF-8, these two symbols are encoded as E4 BD A0 E5 A5 BD₁₆. Find the Unicode code points for these symbols.
8. *Problem:* Translate the 8-bit Two’s Complement number 1101 0101_{2C} into UTF-8 for display.

PART IV

Data Representation

13 Images and Color	131		
13.1 RGB and HSV	132	14.3 Image Masks	150
13.2 HSV to RGB	135	14.4 XOR Cipher	152
13.3 RGB to HSV	137	14.5 Exercises	154
13.4 CMYK	138	15 Error Correcting Codes	155
13.5 Raster Images	138	15.1 Repetition	157
13.6 Alpha Blending	140	15.2 Parity	158
13.7 Vector Images	142	15.3 Berger Code	159
13.8 Exercises	144	15.4 Hamming Code	160
14 Bitwise Operations and Masking	145	15.5 Reed-Muller Code	162
14.1 Flags	146	15.6 Exercises	167
14.2 Bit Masks	149	16 Compression	168

Images and Color

An image is a visual representation of some content; it may contain a variety of shapes and possibly colors. Computer images are divided into two major categories: raster (bitmap), and vector (line) graphics. Most of the content that users traditionally think of as “images” fall into the raster category.

Computer images are also categorized by the type of color they offer. The simplest images may support only black and white, although more commonly many colors are supported. Encoding of transparency and/or translucency is also a factor.

Due to the large amount of data required to represent an image, compression techniques are often applied. Two main forms of image compression are lossless (which always allow the exact original image to be reconstructed) and lossy (which can cause artifacts, blurriness, or discoloration in images, but allow a much smaller file size). Lossless compression is recommended for images that consist of text and sharp edges (like logos or rasterized vector images). Lossy compression is standard for photographs.

Images which are not strictly black and white must have some way of describing what colors are being used. A variety of color models are employed to this end. A color model describes what fundamental properties are used to define colors, and how they mix. A



Raster: image data stored as a rectangular grid of colors.



Vector: image data stored as a collection of shapes.



Color Model: a technique for representing colors using a collection of numeric values.

color model is either additive or subtractive. Color monitors and projectors work by emitting light; by default, the screen is dark. The device emits light which brightens the screen. Such a device works using an additive color model: black is default, and colors can be added and mixed to reach white, which is formed by combining all colors at maximum brightness.

Printed documents, on the other hand, work the opposite way. A piece of paper generally starts out all white by default, and inks or toners in various shades can be added to darken it. If all colors are mixed at their maximum intensity, black is formed.

In any color model, the intensity of each color is usually specified with a number; sometimes floating point numbers 0 through 1 (with 0 being the least intense and 1 being with most intense) are used. More commonly, 8-bit unsigned integers are used, with 0 being the least intense and 255 being the most intense. These values are often written in hexadecimal (00_{16} through FF_{16}).



Pixel: the smallest component of an image that a device displays or prints.

13.1 RGB and HSV

The most common additive color model is RGB, short for Red-Green-Blue. This color model matches the physical construction of most monitors and projectors, which contain a tiny trio of red, green, and blue lights for each pixel which is displayed. On some displays, if you look close enough, you can even see the individual red, green, and blue subpixels. In this additive model, red and green mix to form yellow; red and blue mix to form purple; and green and blue mix to form cyan (a light sea blue). If all three are mixed equally, a shade of gray is formed, with white being created by maximum red, green, and blue.

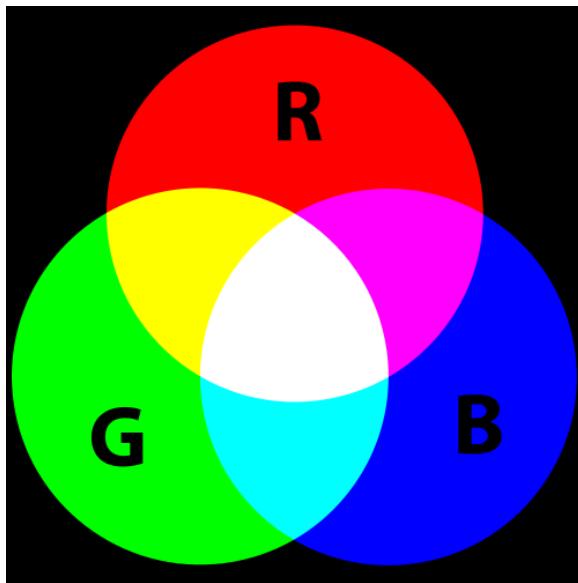
Digital cameras usually also use red, green, and blue sensors. In some cases, sensors for each type of color are packed together. In other cases, one general sensor is used with color filters, and a series of images are captured in quick succession using the different filters. An example of this effect can be noticed with



satellite imagery. In this case, a satellite using red, green, and blue filters (along with a general unfiltered grayscale capture) has taken a picture using four quick exposures. An airplane passing through the field of view was moving fast enough, relative to the exposure speed, to appear in different places in each exposure.



The RGB color model can be visualized as a Venn diagram of the three primary colors. On the outside of the circles, when no color is present, the default is black. In the very middle, when all colors are present, the result is white.



If all three components of an RGB color are the same, or similar, the color is a shade of gray.

RGB colors are often written as a tuple of hexadecimal values, starting with a #. For example, the color specified by #1278DE has a red component of 12₁₆,

a green component of 78_{16} , and a blue component of DE_{16} . This color has mostly blue, with some green and a small amount of red; it appears as a medium blue-green.

Several common colors, and their associated RGB codes, are shown below.

Black	#000000	Gray	#C0C0C0
Red	#FF0000	Green	#00FF00
Blue	#0000FF	Yellow	#FFFF00
Cyan	#00FFFF	Magenta	#FF00FF
White	#FFFFFF	Orange	#FF9900

An alternative color model is HSV, short for Hue-Saturation-Value. HSV defines a color with three numbers, as RGB does, but interprets them very differently. The first number is hue, which defines where on the color spectrum the color falls. The hue maps to a color wheel, and so the range of values for hue is generally 0 through 360 (the degrees around a circle). Red falls at the beginning and end (because hue represents a circle, the beginning and end are actually the same place). Green is centered around 120 degrees, and blue is centered around 240 degrees. Intermediate colors, such as yellow, cyan, or magenta, fall between the centers of the primary colors.

The saturation component relates “how gray” the color is. A saturation of zero gives gray no matter what hue is selected. A medium saturation yields a washed out color. A full saturation (usually on the range of 0 through 100) gives a vivid (but not necessarily bright) color.

The value (sometimes represented as lightness), indicates the intensity of the color. A minimum value will yield black for any combination of saturation and hue; a medium value yields a dark color, and a high value yields a bright color. Notice the distinction between value (which controls brightness) and saturation (which controls vividness). A color with a high value but low saturation will be nearly white, while a color with low value (regardless of saturation) will be nearly



black. A color with both high value and high saturation will be vivid and bright in the selected hue.

The saturation and value are often represented in terms of percentage; with 0% being the lowest and 100% being maximum.

The HSV model, and its related models, are useful for artistic color selection and also for computational analysis of colors, such as computer vision and optical recognition applications.

13.2 HSV to RGB

RGB and HSV are directly convertible, and many color applications will let the user manipulate colors in either or both systems. The basis of this relationship is the division of hue into six categories. These categories describe the “ordering” of the red, green, and blue by intensity. In each category, the primary color is at a fixed, high intensity. The secondary color is either increasing or decreasing as the hue increases.



Recall that hue is represented as a circle, with 360 degrees.

Hue Range	Dominant	Secondary	Secondary Color Is
0 - 60	Red	Green	Increasing
60 - 120	Green	Red	Decreasing
120 - 180	Green	Blue	Increasing
180 - 240	Blue	Green	Decreasing
240 - 300	Blue	Red	Increasing
300 - 360	Red	Blue	Decreasing

For example, consider the RGB color #0571AF. This color is primarily blue (AF_{16}), with a medium amount of green (71_{16}), and a small amount of red. Therefore, the hue for this color would fall in the range of 180 to 240. Likewise, if a color were known to fall in the hue range of 60 to 120, then its RGB components must be ordered $G \geq R \geq B$.

To convert from HSV to RGB, consider H to be the hue in the range 0 to 360, S to be the saturation in the

The “category” conversion process maps the circular hue into a hexagon, causing a slight deformation of the color space. A completely correct translation would apply trigonometric formulas. The error, however, is not much more than 1 degree.

If $S = 0$, then $(R, G, B) = (V, V, V)$

The formulas for the components are often written in a factored form, such as $V(1 - S(1 - F))$ instead of $V - VS + VSF$.

range 0 through 1, and V to be the value in the range 0 through 1. The resulting RGB values will also be in the range 0 through 1. The dominant RGB component, identified by the hue category table, will be the same as V .

Next, we must calculate how far (as a percentage) through the current hue category the color is. This can be found by calculating $F = \frac{H}{60} - \left\lfloor \frac{H}{60} \right\rfloor$.

The secondary RGB component, identified by the hue category table, depends on how the secondary color is changing within that category. If the secondary color is increasing, then the secondary RGB component is defined as $V - VS + VSF$. If the secondary color is decreasing, then the secondary RGB component is defined as $V - VSF$.

Finally, the remaining RGB component is defined as $V - VS$.

■ **Example 13.1** • For example, given a color with hue of 57 degrees, a saturation of 21%, and a value of 34%, what is the RGB representation? First, we identify that this color has a dominant component of red, a secondary component of green, and that the secondary color is increasing. To begin with, $R = 0.34$ (the value). To compute the other two, we first find

$$F = \frac{57}{60} - \left\lfloor \frac{57}{60} \right\rfloor = 0.95 - [0.95] = 0.95 - 0 = 0.95.$$

The secondary component, green, is found with $G = 0.34 - 0.34 * 0.21 + 0.34 * 0.21 * 0.95 = 0.34 - 0.071 + 0.068 = 0.337$. The final component, blue, is found with $B = 0.34 - 0.34 * 0.21 = 0.34 - 0.071 = 0.269$. With ranges of 0 through 1, the RGB for this color is $(0.34, 0.337, 0.269)$.

These can be converted into hexadecimal tuples by multiplying each component by 255 and rounding. In the 8-bit range, the red value is $87_{10} = 57_{16}$, the green value is $86_{10} = 56_{16}$, and the blue value is $69_{10} = 45_{16}$. The hex tuple would be written as #575645. ■



13.3 RGB to HSV

Conversion from RGB to HSV follows the opposite process. Define R, G, and B to be in range of 0 through 1. The resulting S and V values will be 0 through 1, and H will be 0 through 360. Identify the dominant RGB component (simply based on the largest number of the three). Define the value as $V = \max(R, G, B)$ (that is, the value is equal to the dominant RGB component).

Define an intermediate value, D , to be the difference between the largest and smallest RGB components. In other words $D = V - \min(R, G, B)$. With this intermediate value, the saturation can be defined as $S = \frac{D}{V}$.

Finally, the hue can be determined based on the dominant and secondary RGB component. Depending on the secondary component, the value of hue may be negative; to return it to the 0 to 360 range, add 360 to the negative value.

Dominant RGB Component	Formula for Hue
Red	$H = 60 * \frac{G - B}{D}$
Green	$H = 60 * \left(2 + \frac{B - R}{D}\right)$
Blue	$H = 60 * \left(4 + \frac{R - G}{D}\right)$

■ **Example 13.2** • For example, given the color #23375F, what is the HSV representation? Translate these values into the 0 through 1 range. For red, $23_{16} = 35_{10}$; for green $37_{16} = 55_{10}$, and for blue $5F_{16} = 95_{10}$. To find the translated range, divide each decimal value by the 255. In the translated range (rounded to three decimal places), the tuple is $(0.137, 0.216, 0.373)$.

The dominant component becomes the value, so $V = 0.373$. Next, find $D = V - \min(R, G, B) = 0.373 - 0.137 = 0.236$. Calculate the saturation $S = \frac{D}{V} = \frac{0.236}{0.373} = 0.633$.

$\sum_{n=1}^{\infty} \frac{1}{n^3}$ In the event of a perfect shade of gray, where each value R , G , and B are equal, then $D = 0$.

$\sum_{n=1}^{\infty} \frac{1}{n^3}$ If $V = 0$, then let $S = 0$ and $H = 0$ as well.

The value for hue in particular is impacted by the repeated rounding of results. When this algorithm is executed by computer, higher precision is used and the rounding errors are not significant.

Finally, determine the hue. The dominant RGB component is blue, so apply the formula $H = 60 * \left(4 + \frac{R - G}{D}\right) = 60 * \left(4 + \frac{0.127 - 0.216}{0.236}\right) = 60 * (4 + -0.377) = 217$. There, this color would be represented with a hue of 217 degrees, 63% saturation, and 37% value. ■

13.4 CMYK

CMYK is a subtractive color model used primarily for printing. Unlike RGB, where the absence of color results in black, and all colors result in white; in CMYK, the absence of any color (ink) results in a white page, and the presence of all colors forms black. Strictly speaking, C (cyan), M (magenta), and Y (yellow) alone are enough to form any color, however, to save substantial cost, black ink is used and then colorized with the minimum amount of colored ink necessary.

No direct conversion between CMYK and RGB exists without defining the precise tone relationship of the colors in use. Approximations exist, and most publishing software will allow a printer to define the transformation so that an image is colorized on screen and on paper as similarly as possible.

13.5 Raster Images

A raster image is constructed of a rectangular grid of pixels. Each pixel represents a single color, the range of which depends on the capability of the image. Color range (often called color depth) is normally expressed in terms of bits per pixel. The number of bits per pixel expresses how many colors can be represented by that pixel. Exactly what color each bit sequence refers to depends on the particular image encoding technique. Common values for bits per pixel are 1 (black and white only), 8 (256 colors), 24, and 32 bits.

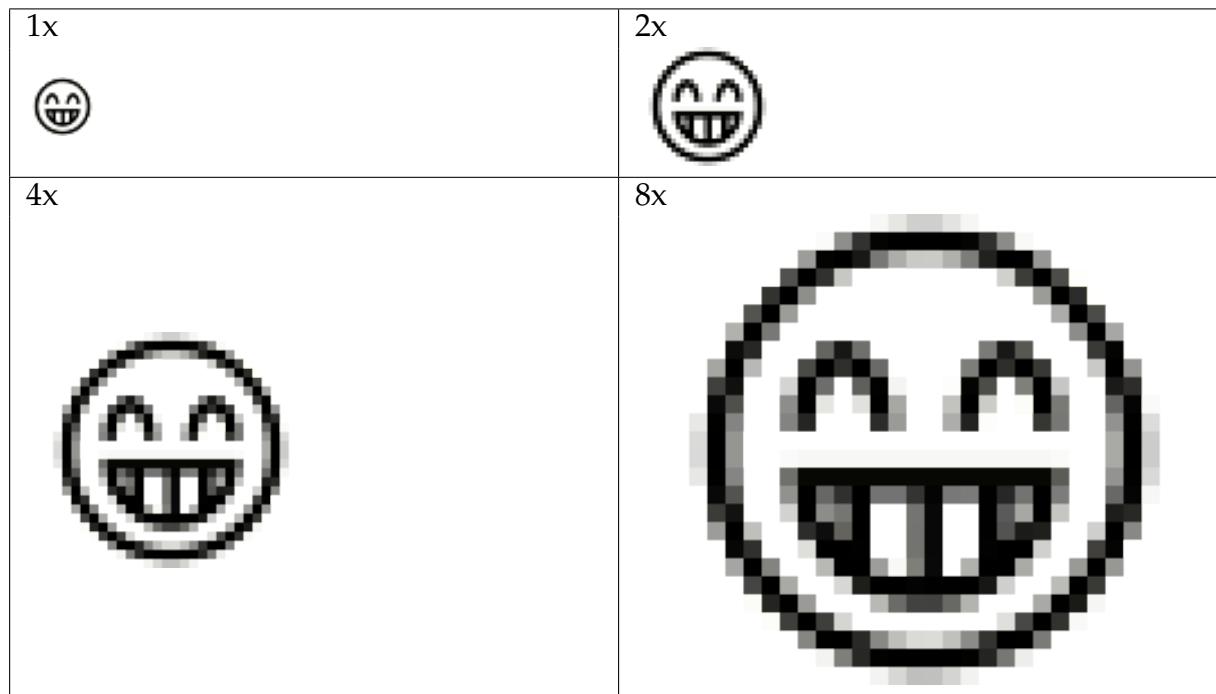


Color Depth: bits per pixel in an image.

Due to the raster image's internal representation as a grid of pixels, raster images suffer when zoomed. To facilitate zooming, many images are produced with a very large number of pixels, and then zoomed out for default display. In addition, some smoothing techniques exist to make pixel edges blend. In all cases, however, zooming on a raster image eventually leads to blurriness and a loss of detail.

The raster image below and left is defined by 32 pixels horizontally and 26 pixels vertically, for a total of 832 pixels. In the sequence below, this image is zoomed repeatedly to reveal the lack of detail.

In general, an uncompressed image of $w \times h$ pixels and a color depth of n bits will require about $\sum_{n=1}^{\infty} \frac{1}{n^8}$ bytes of storage. The first three bytes store the width, height, and color depth.



To store an image, the image height and width (in number of pixels), and the color depth (bits per pixel) must be stored along with the image. If the bits per pixel is not divisible by 8, then the pixels may or may not be packed. Unpacked pixels are expanded to meet the byte boundary; this takes more space but is easier for programs to process.

Images with a color depth of 1 bit are usually represented with the two colors of black and white; although in theory any two colors could be used. Images of this type are used in black and white printing, and for cer-

 Packed 1-bit raster images will require about $2 + \frac{wh}{8}$ bytes of storage.



Palette: a selection of colors available for use in an image.



GIF: Graphics Interchange Format, an 8-bit lossless compressed image format popular for small animations.

tain display techniques, such as image masking. Storage on disk and in memory is generally done at the byte (8 bit) level. Unpacked bitmaps will use 00_{16} for black and 01_{16} for white, but this representation is very inefficient. Packed bitmaps will represent each pixel with one bit, 8 pixels per byte.

Images with a color depth of 8 bits fall into one of two categories: grayscale or color. In either case, 8 bits (1 byte) of data represents one pixel. Grayscale images support 256 shades of gray, from 00_{16} being black and FF_{16} being white. A basic grayscale image can be generated from a color image by considering only the value of the color image represented in HSV. However, this translation does not consider the differences of hue in human vision, and so does not have the best possible dynamic expressiveness.

Eight bit color images often use a palette to indicate the value of each color. For example, in GIF images, the 256 color palette is first defined with 24 bit RGB per color. These colors are then referenced in the pixel data of the image. The use of a palette allows image software to optimize the number of colors available where the most color expressiveness is needed, at the expense of unused colors. For example, a picture of an ocean might have a palette with many blues and greens, and few reds. GIF images also allow one of the colors in the palette to be marked “transparent”, which indicates that no pixel should be drawn at locations with that color. Transparency enables the effect that makes some images appear to be non-rectangular.

13.6 Alpha Blending

Most images are stored with either 24 or 32 bits of color depth. The 24 bit color depth associates an RGB values (8 bits for each of red, green, and blue) with each pixel. Extending this to 32 bits usually does not change the number of bits associated with RGB, but adds a forth channel for translucency, called the alpha channel. Unlike the simple transparency of GIF, an alpha channel allows for varying translucency, so that portions of an

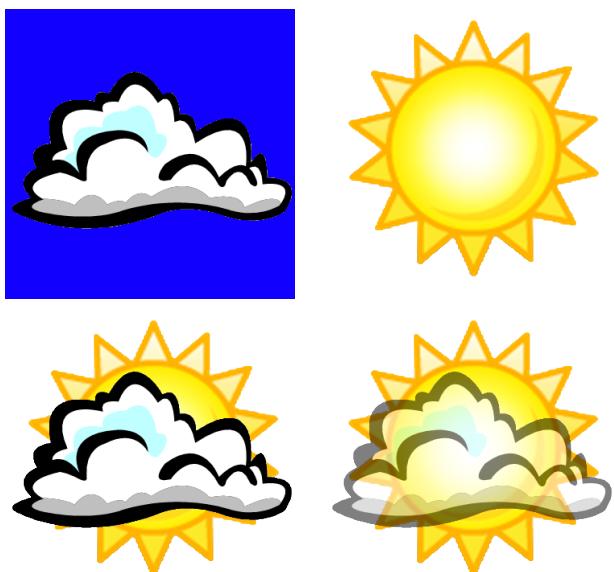


image may partially show what is behind them.

In order to determine what color pixel to actually display, the computer must blend the translucent image with the background. For these formulas, the alpha will be assumed to be in range of 0 through 1.

To blend two colors with alpha channels, let the top color be defined as (R_0, G_0, B_0, A_0) and the bottom color be defined as (R_1, G_1, B_1, A_1) . In that case, first define the alpha channel for the new color as $A_2 = A_0 + (1 - A_0)A_1$. Next, the components of the new color (R_2, G_2, B_2) can be defined as $R_2 = \frac{A_0 R_0 + (1 - A_0)(A_1 R_1)}{A_2}$, and likewise for G_2 and B_2 .

Consider first an image of a cloud, with a blue background (left), and an image of the Sun, with a white background (right). Neither of these images has transparency. On the second row (left), we replace the blue background of the clouds with transparency, and overlay the cloud image on top of the sun image. Both images are still rectangular, and have the same shape; however, the transparent portions of the cloud image admit the pixels from the sun image behind it. Finally (bottom right), we apply an alpha channel to the remaining non-transparent portion of cloud. The alpha channel is set to 50%, and the alpha blending effect is clearly seen.



Alpha: a component of a color indicating how translucent it is.



Alpha Blending: mixing a translucent color over a background to determine the actual color to display at that location.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

In the case that the background color (R_1, G_1, B_1) has no alpha channel, a translucent foreground color (R_0, G_0, B_0, A_0) can be blended with it with the simplified formula $R_2 = A_0 R_0 + (1 - A_0)R_1$, and likewise for G_2 and B_2 .



Translucent colors may be represented with ARGB or RGBA or some other arrangement of values; the sequence of letters indicates the order in the hexadecimal code in which each component appears.

Colors with an alpha channel may be represented as ARGB, which adds 2 hexadecimal digits to the RGB color code, indicating the alpha value. An alpha value of 00_{16} means fully transparent (0%), and an alpha value of FF_{16} means fully opaque (100%). Thus, the color `#80FFFF00` is a half-transparent yellow.

13.7 Vector Images

Vector images are constructed using a collection of shape primitives. Each primitive, such as a rectangle or circle, can be assigned styles and drawn at a particular position and size. Vector images are constructed using a language which describes to the computer what shapes should appear where. The computer can then render these shapes into a raster image of the appropriate size. If the user zooms in on a vector image, the computer re-renders it at the new size, creating a smooth appearance.

One language used to describe vector images is Scalable Vector Graphics (SVG). SVG is stored on disk as a sequence of characters, and these characters are interpreted to instruct the computer to draw some shape. The SVG file below, for example, describes a rounded rectangle with a yellow filling and black edge.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">

<rect x="20" y="20" rx="20" ry="20" width="250" height="100"
style="fill:yellow;stroke:black;stroke-width:5;opacity:0.5"/>

</svg>
```

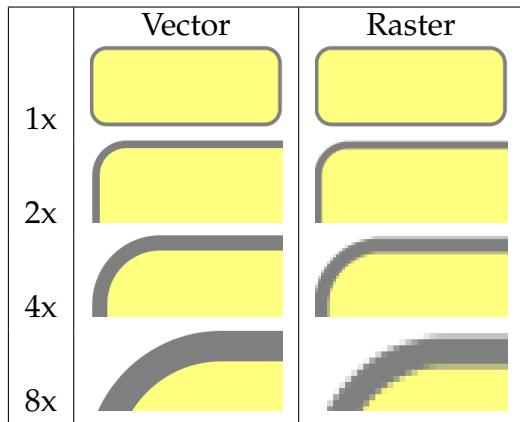
The table below shows the result of zooming in on the upper-left corner of this rectangle. In the vector version, the computer re-renders the rectangle at each zoom, resulting in no loss of detail. On the other hand,



SVG is an example of a general syntax called XML, usually stored using UTF-8.



if the original rectangle is first converted into a raster image 256 pixels wide and 110 pixels high, and then the same zoom sequence is undertaken, pixelation appears quickly.



13.8 Exercises

Solutions to these exercises can be found in Appendix A.13 on page 317.

1. *Problem:* Describe the colors represented by the following RGB codes.
 - (a) #4682B4
 - (b) #C71585
 - (c) #A52A29
 - (d) #807E82
 - (e) #98FB99
2. *Problem:* Describe the colors represented by the following HSV values.
 - (a) 260 degrees, 35% saturation, 94% value
 - (b) 134 degrees, 65% saturation, 41% value
 - (c) 46 degrees, 10% saturation, 95% value
 - (d) 321 degrees, 10% saturation, 9% value
 - (e) 321 degrees, 90% saturation, 9% value
3. *Problem:* Convert the HSV color with hue of 200 degrees, a saturation of 50%, and a value of 85% to RGB.
4. *Problem:* Convert the RGB color #274F07 into HSV.
5. *Problem:* A large uncompressed, unpacked black and white (1 bit) raster image requires about 100,000 bytes. If the image is instead stored as uncompressed, *packed* black and white (1 bit), about how much space will be required?
6. *Problem:* An uncompressed 24-bit color raster image requires about 200,000 bytes. If an 8-bit alpha channel is added to the image, about how much space will the new image require?
7. *Problem:* An opaque background of color #3409AB is overlaid with the translucent ARGB color #0F4A7A99. What is the resulting RGB display color?
8. *Problem:* An ARGB color #B1A00591 is placed on top of another ARGB color #55667788. The two colors are alpha blended; what is the resulting ARGB color?



Bitwise Operations and Masking

The Boolean operators, such as AND, OR, and NOT, can be extended beyond functioning on single bits to work on bit sequences. When Boolean operators are applied to bit sequences, they are called bitwise operators. The input and output values are usually represented as unsigned integers. Bitwise operations have many applications in systems programming, cryptography, and networking.

A bitwise operation is performed by aligning two binary numbers of equal length and for each position performing the operation to produce a result number. Commonly `&` is used for AND, `|` is used for OR, `~` is used for NOT (also called complement), and `⊕` or `^` is used for exclusive-OR. In the following example, the symbol `&` refers to the bitwise operator AND. Note that the AND operation is applied to each column (position) independently; there is no carrying or borrowing that ever occurs in bitwise operations. Each position is independent of the others.

$$\begin{array}{r} 0011_2 \\ \& 1010_2 \\ \hline 0010_2 \end{array}$$

Although bitwise operations always occur at the bi-



By showing all possible inputs, as in this example, a bitwise operation can act like a truth table.

nary level, there is no reason the inputs must be written that way. The previous example might also be written as $3_{10} \& 10_{10} = 2_{10}$.

The NOT operator inverts all bits in a number. The number of bits available must be known.

■ **Example 14.1** • For example (six bits) $\sim 101001_2 = 010110_2$. The same operation in other bases assumes a conversion to and from binary first. For example, (eight bits) $\sim FA_{16} = 05_{16}$, and (seven bits) $\sim 123_{10} = 4_{10}$. If eight bits are used, then $\sim 123_{10} = 132_{10}$. The reason for the distinction is that in seven bits, $123_{10} = 111\ 1011_2$, and $\sim 111\ 1011_2 = 000\ 0100_2 = 4_{10}$. In eight bits, $123_{10} = 0111\ 1011_2$, and $\sim 0111\ 1011_2 = 1000\ 0100_2 = 132_{10}$. ■

The toggling property of exclusive-OR gives it two interesting identities: $a \oplus a = 0$, and from that it follows that $a \oplus b \oplus b = a$.

The OR operator functions in a similar manner to the AND operator, except that the resulting position will have a 1 if either input position had a 1. For example, $110\ 0101_2 | 010\ 1010_2 = 110\ 1111_2$. The resulting value of an exclusive-OR is a 1 in each position where either but not both of the source values had a 1. For example $1101_2 \oplus 0101_2 = 1000_2$.

Bitwise operations permit compact efficient utilization of individual bits in a number, and tend to be extremely fast for the processor to perform (due to the lack of any borrowing, carrying, or other special cases). Thus, they are most appropriate when space and time are at a premium, such as embedded devices or high performance computing.

14.1 Flags

In many situations, a substantial number of Boolean parameters may be available to some influence how a computation is performed. These parameters can describe options, settings, or any other configuration data. Rather than store each Boolean value in a separate area, they can be combined using flags. To use a flag, a single integer number is used as a storage space for many Boolean values (up to the number of bits in the



integer).

■ **Example 14.2** • Imagine we want to store which days of the week (Sunday through Saturday) a person is available to work. A variety of inefficient storage mechanisms are possible, such as a variable length string containing a letter for each available day, or a fixed length 7 byte string with a series of Y/N values, one for each day. However, a complete accounting of available or unavailable for each day can be stored in 1 byte per person. Instead of thinking of the byte (8 bits) as an integer number, consider each bit as a yes or no setting for a particular value, in this case one day. Only seven days exist, so only seven bits are needed, although most computers allocate bits in a minimum of 8 bit blocks.

Each value will be assigned one position in the number. In this case, we'll let Sunday be the leftmost bit of 7 and Saturday be the rightmost bit. This gives the values:

Day	Value in Binary	Value in Hex
Sunday	100 0000 ₂	40 ₁₆
Monday	010 0000 ₂	20 ₁₆
Tuesday	001 0000 ₂	10 ₁₆
Wednesday	000 1000 ₂	8 ₁₆
Thursday	000 0100 ₂	4 ₁₆
Friday	000 0010 ₂	2 ₁₆
Saturday	000 0001 ₂	1 ₁₆

If a certain person is available Monday, Wednesday, and Saturday, their value would be 010 1001₂ = 29₁₆. Note that the 1 bits correspond to each available day. A person who was not available on any day would have a value of 000 0000₂ = 0₁₆; likewise, a person available every day of the week would have the value 111 1111₂ = 7F₁₆. ■

Bitwise operations permit efficient checking of individual flag values in a number, as well as setting, clearing, and toggling of flag values. To check if a particular flag with value f is set in a number n , determine if $n \& f = f$. To set a particular flag with value f in a number n , regardless of whether or not it is already set,

 *Flag:* a bit which indicates the presence or absence of a particular condition or setting.

 A series of flags will all be exact powers of two, so that each flag represents a single bit position.

 The value for a certain combination of flags can be found by adding, or by bitwise OR'ing, all the desired flag values.

 If unsigned numbers are known to be in use, the flag check can be done slightly faster by checking for $n \& f > 0$, as most processors have a quick, built-in check for zero equality.



update $n = n \mid f$. To clear a particular flag with value f in a number n , regardless of whether or not it is already set, update $n = n \& \sim f$. Note that two operations are involved in this step.

■ **Example 14.3** • For example, using the previous example, is a person with value 33_{16} available on Tuesday? The flag value for Tuesday is $001\ 0000_2$; translating 33_{16} into binary gives $011\ 0011_2$. We find that $011\ 0011_2 \& 001\ 0000_2 = 001\ 0000_2$. This value is greater than zero, so the person is available on Tuesday.

Taking the person from the previous example, we want to show that they are available Tuesday and Wednesday (in addition to any days they are already available). The original value is $011\ 0011_2$, Tuesday is represented as $001\ 0000_2$ and Wednesday as $000\ 1000_2$. Note that $011\ 0011_2 \mid 001\ 0000 = 001\ 0011_2$, no change, because the Tuesday flag was already set. Next, $011\ 0011_2 \mid 000\ 1000 = 011\ 1011_2$, which now shows the Wednesday flag being set.

Now imagine we have been informed that the person is no longer available on Tuesday. Their current value is $011\ 1011_2$ and Tuesday has the flag value $001\ 0000_2$. If we want to clear this flag, we can apply $011\ 1011_2 \& \sim 001\ 0000_2 = 011\ 1001_2 \& 110\ 1111_2 = 010\ 1001_2$. The NOT operator first inverts the flag, and the AND operator ensures the lone zero bit clears out that position, while the other one bits leave the other positions untouched. ■

A particular flag f can be toggled in a number n a single step using exclusive-OR; update $n = n \oplus f$ to toggle the flag. This step works because the zeros in the flag, when used with exclusive-OR, will leave the original bits alone; the 1 bit in the flag position will cause the input bit to be toggled.

Anytime a series of options is represented using powers of two, you can assume bitwise flags are being used. Flag operations can be summarized in this table, where f is the flag value (a power of two), and n is the number containing the flags.

 Many bitwise operations depend on the Boolean algebra rules. For example, the clearing of a flag uses the rules $x \wedge T = x$ as well as $x \wedge F = F$ to keep all values except the one to clear.

 Any list of Boolean values can potentially be condensed using bitwise operations as long as the size of the list is fixed and not greater than the number of bits in your programming language's integer.



Operation	Technique
Set a Flag	Update $n = n \mid f$
Clear a Flag	Update $n = n \& \sim f$
Toggle a Flag	Update $n = n \oplus f$
Check a Flag	Determine if $n \& f = f$

14.2 Bit Masks

A bit mask is a technique to describe which portions of a number should be considered in some operation. In particular, masks are often used to define a range of integers, especially in networking. In the IPv4 networking protocol, each computer has a 32-bit identifier (an IP address). To describe which computers are directly on the local network, a subnet mask is used. IPv4 addresses are written using four blocks, where each block is a base-10 integer 0 through 255. Thus, the IP address 1.2.3.255 corresponds to number $010203FF_{16}$.

■ **Example 14.4** • Consider a particular computer which has an IP address of 10.50.1.170 ($0A3201AA_{16}$) and a netmask of 255.255.255.224 ($FFFFF000_{16}$). The zeros in the netmask indicate bits that may be used to number computers on the local network. Thus, this network can have at most

$$\sim FFFF000_{16} =$$

$$0000001F_{16} =$$

31_{10} different computers. The IP address range starts at

$$0A3201AA_{16} \& FFFF000_{16} =$$

$$0A3201A0_{16} \text{ (10.50.1.160) and continues up through}$$

$$0A3201AA_{16} \mid \sim FFFF000_{16} =$$

$$0A3201AA_{16} \mid 0000001F_{16} =$$

$$0A3201BF_{16} \text{ (10.50.1.191).} \blacksquare$$

Note that the minimum value of a bitmask is found using the AND operator (essentially, to clear all settable bits) and the maximum value of a bitmask is found using OR and NOT (essentially, to set all settable bits).

A similar technique is used for firewalls to limit access to network resources to only approved portions of the network. For example, a corporate payroll server might be limited to only be accessible from comput-

 **Bit Mask:** a series of bits that are manipulated using bitwise operations, usually to define portions of number as usable or not.



Bitmasks are often abbreviated using slash notation, where $/n$ means n 1s followed by the rest being 0s. For example, the 32-bit mask $/20$ is $FFFF0000_{16}$ (20 1s followed by 12 0s).

ers in the accounting department. An access control system can use a bitmask to screen incoming requests based on the IP address of the requesting computer to quickly see if it falls into the range of allowed computers.

Very easy and fast operations are required to enable hardware, such as firewalls and routers, to handle large amounts of network traffic. Bitwise operations were chosen to allow network zones to be defined and still be processed extremely fast, ensuring high network throughput.

IPv6 networks (composed of 128 bit addresses compared to 32 bits for IPv4 networks) use a prefix length instead of a netmask, however, the mechanism works essentially the same way.

14.3 Image Masks

To create the appearance of transparency in images while avoiding the computational expense of alpha blending, image masks may be used. Image masks provide an extremely fast technique for simulating transparency in raster images.



The technique of layering raster images using bitwise operations is often called a “bit blit”, short for bit block transfer.

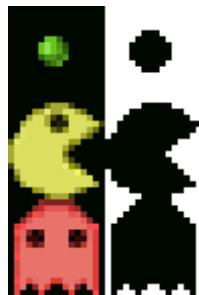
The general technique involves the source image and a 1-bit “cut out” which defines the transparent regions. The portions which are to remain background have a value of 1, and the portions which are to become the new image have a value of 0. The portions of the new image which should be transparent also are represented with the value 0.

The cut out is applied against the background using the AND operator (so that portions that will contain the new image become all 0s, and other portions remain untouched), and then the new image is applied using the OR operator.

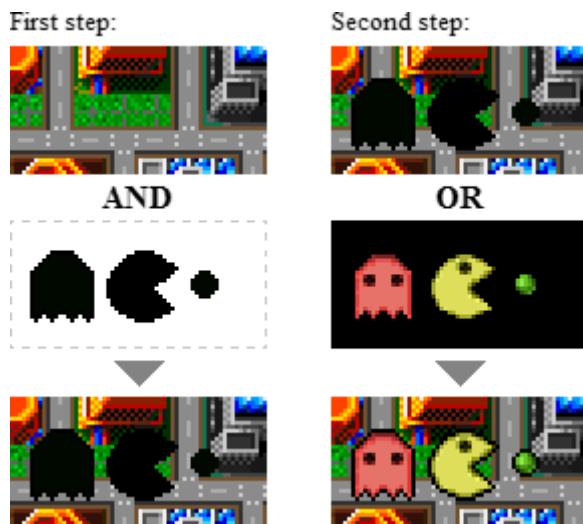
■ **Example 14.5** • Imagine we want to render some characters onto a background. The characters could be defined, along with a mask, to allow quick transparent



rendering. Consider the three characters below (left) and their associated image masks (right).



The characters may be rendered onto a background by first applying the image mask with the AND operator, and then applying the character image with the OR operator.

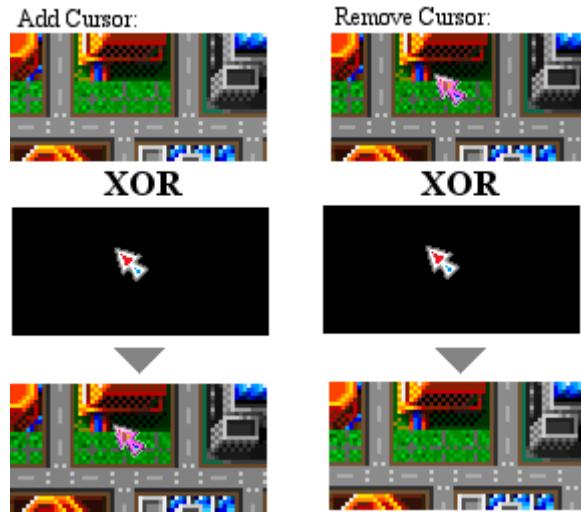


An even faster approach, although one that causes loss of color continuity, is simply to exclusive-OR the background image with the foreground image, and then remove the foreground image later with another exclusive-OR. This technique can be used for mouse cursors and other transient images that need only draw user attention to certain area of the screen. The exclusive-OR technique is also useful if an original copy of the background is unavailable. In the previous

technique, a portion of the background is cut out and removed; using exclusive-OR, the original background can be restored.

■ **Example 14.6** • For example, if a background pixel has the color #12AB90 (greenish-blue) and the cursor pixel with color #888888 (gray) is applied with an exclusive-OR, then the new pixel will be rendered with color #9A2318 (red; very distinct from the original color). To remove the cursor, simply apply #9A2318 exclusive-OR with the cursor #888888 and the original pixel color will be restored. As the cursor moves across the background, its color composition will change. ■

The sequence belows show a cursor placed and then removed using exclusive-OR.



14.4 XOR Cipher

Bitwise operations play a fundamental role in cryptography as well. The exclusive-OR cipher, by itself, is merely a curiosity, but it appears as a core component of almost all modern encryption techniques. The exclusive-OR cipher is an example of a symmetric cipher, meaning that the same key is used to perform both encryption and decryption.



To apply the exclusive-OR cipher, a plaintext and a passcode byte sequence are both required. The plaintext represents the content (which need not represent text) to encrypt, and the passcode is the byte sequence to encrypt and later decrypt the content with.

Each byte of the plaintext is exclusive-OR'd with a corresponding byte from the passcode. Normally, the passcode is substantially shorter than the plaintext, so the passcode is repeated as many times as needed. The result of the exclusive-OR is the ciphertext, the encrypted content. This content can be decrypted by applying the same exclusive-OR again with the passcode.

■ **Example 14.7** • For example, imagine we want to encrypt the byte sequence $5A B2 07 D2 EE_{16}$ using the passcode $1A 2B_{16}$. Align each byte of the plaintext with the corresponding passcode byte, repeating the passcode as needed to provide for the entire length.

$$\begin{array}{r} 5A \quad B2 \quad 07 \quad D2 \quad EE \\ \oplus \quad 1A \quad 2B \quad 1A \quad 2B \quad 1A \end{array}$$

For each pair of plaintext and passcode bytes, convert them into binary and apply the bitwise operation exclusive-OR to each bit.

$$\begin{array}{rcl} 5A_{16} & = & 01011010_2 \\ 1A_{16} & = & 00011010_2 \\ \hline & & 01000000_2 = 40_{16} \end{array}$$

Likewise for all remaining sequence. The encrypted content is $40991DF9F4_{16}$. To decrypt this content, apply the same procedure with the same passcode. Again, the first byte is illustrated here, and the rest follow likewise.

$$\begin{array}{rcl} 40_{16} & = & 01000000_2 \\ 1A_{16} & = & 00011010_2 \\ \hline & & 01011010_2 = 5A_{16} \end{array}$$



The concept of XOR cipher works precisely the same as the XOR sprite overlay.



The plaintext could represent an image, text in some encoding (such as UTF-8), music, or any other content.



Exclusive-OR, like all bitwise operations, can be applied equally well to any length of bits; it is convenient for our purposes to divide the streams into bytes, but this does not affect the result.

14.5 Exercises

Solutions to these exercises can be found in Appendix A.14 on page 322.

1. *Problem:* Perform the following bitwise operations on 8 bit numbers:
 - (a) $5A_{16} \& 99_{16}$
 - (b) $3B_{16} | 27_{16}$
 - (c) $4F_{16} \oplus 31_{16}$
 - (d) $\sim 5F_{16}$
3. *Problem:* Determine which types of imagery are available given the following values.
 - (a) 11_{10}
 - (b) 11_8
 - (c) F_{16}
 - (d) 0110_2
4. *Problem:* How many devices can participate in an IPv4 network with a netmask of 255.255.255.240? (Excluding the use of any gateways, etc.)
5. *Problem:* If a device has an IPv4 address of 10.5.77.203, and a netmask of 255.255.255.252, what is the range of IPv4 addresses it can directly communicate with?
6. *Problem:* A pixel of color #45BC09 is exclusive-OR'd with a background pixel of color #045A9F. What is the resulting color?
7. *Problem:* Using the XOR cipher, encrypt the UTF-8 phrase "Hello World" with the passcode "aBc".
8. *Problem:* Using the XOR cipher, decrypt the content represented by $3A100E0F10091D_{16}$ using the UTF-8 passcode "xyz" and show the output using UTF-8.

Image Type	Flag Value (base 10)
B & W Photo	1
Topographic Map	2
Shaded Relief	4
Color Photo	8

2. *Problem:* For each of the following, find the appropriate flag value.
 - (a) The area has shaded relief and color photo available.
 - (b) The area has black and white photo, as well as color photo, available.
 - (c) The area has all four types of imagery available.
 - (d) No imagery is available for the area.

Error Correcting Codes

Connections between computing devices transfer more data today than any time in history. The rise of mobile devices with Internet access means that large quantities of data are being transmitted at faster and faster speeds over narrower channels with increasing probability of interference. Whether wired or wireless, physical anomalies can interfere with data transmission and introduce errors.

Ultimately, all data is transmitted as bits. An error occurs when a bit is received as the opposite value from what was sent; for example, if I send you a 1 and you receive a 0, or vice versa. A transmission medium which is subject to errors is known as a noisy channel. In order to preserve data intact across a possibly noisy channel, computer scientists have developed many techniques for detecting and correcting errors.

If one or more errors is detected with the benefit of a code, the sender can be asked to retransmit the block until no errors occur. This process interrupts the normal transmission and slows down the throughput of the channel. Some codes allow the receiver to correct one or more errors without asking the sender to retransmit. These codes allow the sender to maintain a higher throughput. In all cases, every error detecting or correcting code has a limit; errors beyond the limit may slip past undetected. In order to safeguard data transmissions, multiple methods of error detection are

Error Correcting Code: a technique for encoding a sequence of bits such that certain kinds of transmission errors can be detected, and in some cases, automatically corrected.





Burst Error: a contiguous sequence of bits all of which are incorrect.



Unidirectional Error: an error that can only occur in one direction; such as flipping a one to a zero, but not the other way around.



Hamming Distance: given two bit sequences of equal length, the number of positions in which the values differ.



Code Rate: a measure of the overhead of an error detection or correction code.

applied at different levels to make an undetected error very unlikely.

Many error correcting codes assume that errors occur randomly and without bias (that is, the error is just as likely to flip a 0 to 1 as it is to flip a 1 to a 0). However, not all physical noisy channels share this trait. In particular, the channel may have the property of burst errors and/or unidirectional errors.

A channel prone to burst errors may be usually correct, but when errors occur, long and entire sequences are found in error. For example, an incorrectly shielded wire may accurately conduct bits except when a neighboring wire is powered on, in which case interference ruins the transmission property of the first wire.

Some channels may be prone to unidirectional errors. A channel which experiences unidirectional errors only has errors that flip bits in one direction, usually one to zero, but not zero to one. Such errors might occur in the situation of a long distance wireless connection where signal attenuation causes loss of bits occasionally. Unidirectional errors are easier to detect and correct than unbiased errors because the 1s can be assumed to be correct, and only the 0s may be in errors.

In order to evaluate various error detecting and correcting codes, several measures were created. The Hamming distance is useful for counting the number of errors that have occurred in a particular sequence of bits. Two sequences have a Hamming distance of 1 if one error occurred, a distance of 2 if two errors occurred, and so on.

All error detection or correction codes work by adding additional bits to the data sequence to be sent. Some codes require significantly more overhead than others. The code rate of an error detection or correction code is defined as the ratio of message data bits to total bits. For example, if a certain code transmits 8 data bits along with 2 error detection bits, that would be a (10,8) code.

When counting the number of errors detected and/or corrected by these codes, it is conventional to assume

the worst case scenario.

15.1 Repetition

The simplest of all codes is to repeat the message several times. Repetition is easy to implement and understand, but very inefficient (low code rate). Repetition can be categorized primarily by how many times each message packet will be sent; if each message is sent twice, one error can be detected and no errors can be corrected (we cannot determine which of the two messages is the correct one).

The reason only one error can be detected in the worst case is that two errors could potentially align in the same position in each copy. Consider the message 0101_2 , sent twice using repetition: $0101\ 0101_2$. Two errors could occur as follows: $010\boxed{0}\ 010\boxed{0}_2$. In this case, both copies appear identical to the receiver, and the error passes undetected.

If the message is sent three times, up to two errors can be detected, or up to one error can be corrected. The error correction works by majority vote; if a position is found to be inconsistent, whichever value is most common in that position is taken as the correct value. If only one error has occurred, this will always yield the correct original data.

The risk of error correction in the above case is that two errors, while detected, could be incorrectly “fixed” by a wrong vote.

If the message is sent more than three times, additional errors can be detected and corrected. In general, if the message is sent k times, up to $k - 1$ errors can be detected, or $k - 2$ errors can be corrected.

Repetition’s low code rate combined with its unspectacular error detection rate mean that this technique is essentially never used in practice.

Although the error detection count is based on worst case scenario, you might notice that repetition with three or more instances is well suited to correct against burst errors.

If the message length is n , and the number of instances is k , then the repetition code has a code rate of $R(kn, n)$.

15.2 Parity



Parity Bit: a bit making the number of 1 bits in a message even or odd, as selected.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

If the message length is n , then the parity code has a code rate of $P(n + 1, n)$.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

Let the bits of a message be b_1, b_2, \dots, b_n . The value of an even parity bit can be computed as $P_e = b_1 \oplus b_2 \oplus \dots \oplus b_n$.

A simple and common error detection code is the parity bit. The parity bit is a single bit attached to a message indicating how many 0s or 1s are present in the message. Parity is referred to as either even or odd, referring to the total number of 1 bits in the message after the parity is added.

For example, using even parity, the parity bit (0 or 1) will be added to the message to make the total number of 1s in the message even. If the message is 0010_2 , then the parity bit will be 1, making the final coded messages $0010\ 1_2$, which has an even number of 1s (counting both data and parity). Likewise, if the message is 1100_2 , the parity bit will be 0, ensuring the total number of 1s remains even. The final coded message would be $1100\ 0_2$.

■ **Example 15.1** • Assume the message 1010_2 will be transmitted with even parity over a noisy channel. Here, the parity bit is determined to be 0 (because there are already an even number of 1s), and the message $1010\ 0_2$ is transmitted. The receiver receives $1\ 1\ 10\ 0_2$. Counting the number of 1s, the receiver finds it is odd, and knows an error has occurred. Two errors, however, would be undetectable: $0\ 1\ 10\ 0_2$ has an even number of 1s, which is as the receiver expects, and the errors pass undetected. ■

A parity bit can detect only one error, and correct none. However, it has the advantage of being very lightweight and easy to compute. A logic circuit can compute a parity bit using only XOR gates.

An extended version of parity which uses multiple bits by counting the number of 1s is the checksum. Although this technique decreases the chance of an undetected two-bit error, it is still only guaranteed to detect one error.



15.3 Berger Code

The Berger code is an error detecting code designed for channels with unidirectional errors which may only convert 1s to 0s. The Berger code works by counting the number of 0 bits in a piece of data. This count is attached to the end of the data as an unsigned integer of length $\lceil \log_2(n + 1) \rceil$ where n is the number of data bits. The receiver counts the number of 0s and compares it to the count specified by the code. Errors in the data section will drive the number of 0s up, while errors in the check section will drive the desired number of 0s down; thus, the only way the number of 0s in the data and the number of 0s indicated by the count will match is if no errors occur.

The Berger code is defeated if a 0 to 1 style error is possible; it is also unable to specify an error correction (what bit the error(s) occurred in, specifically), thus a detected error requires a retransmission from the sender.

■ **Example 15.2** • For example, consider the data $0010\ 1101\ 1010_2$ which is to be transmitted across a channel subject to unidirectional error. In this case, a Berger code may be applied. Twelve bits of data are being transmitted, so $\lceil \log_2(12 + 1) \rceil = 4$ bits of code will be used. There are six zero bits in the data, thus the check portion is 0110_2 . The sender transmits the data followed by the check $0010\ 1101\ 1010\ 0110_2$.

We will consider four cases: no errors, errors in the data only, errors in the check only, or errors in both. If no errors occur, the receiver receives $0010\ 1101\ 1010\ 0110_2$. It separates out the data from the check, counts the number of zeros in the data and confirms that this matches the check. The data is accepted as error-free.

Assume unidirectional errors in the data occur; the receiver receives $0010\ 1\textcolor{orange}{0}\ 01\ 10\ \textcolor{orange}{0}0\ 0110_2$. In this case, when the receiver counts the number of zero bits in the data, the total comes to eight; but the check portion specifies there should only be six. The receiver discards the data and asks the sender to resend it.



Berger Code: error detecting code capable of detecting any number of errors as long as all errors are of one type (such as 1 to 0).



The receiver and sender must already be in agreement over what code to use and how many bits the code is being applied to.

Next, consider the case of unidirectional errors in the check; the receiver receives $0010\ 1101\ 1010\ 01\ \underline{0}\ 0_2$. In this case, when the receiver counts the number of zero bits in the data, the total comes to six; but the check portion specifies there should only be four. Although the data is actually intact, the receiver cannot determine if the error is isolated to the check bits or not, and must ask the sender to resend.

Finally, consider the case of unidirectional errors in both the data and the check; the receiver receives $0010\ \underline{0}\ \underline{0}\ 01\ 1010\ \underline{0}\ \underline{0}\ 10_2$. In this case, when the receiver counts the number of zero bits in the data, the total comes to eight, but the check portion specifies there should only be two. The receiver discards the data and asks the sender to resend it. ■



Hamming Code: a simple error correcting code that can correct one error.

15.4 Hamming Code

Error detection codes suffer from the downside that when an error is detected, the channel must be interrupted in order to ask the sender to resend the damaged message again. Such an interruption can dramatically slow down the apparent throughput of a channel. As much as possible, such interruptions should be avoided.

In order to avoid interrupting the sender, it is possible to develop codes which not only detect but can also correct certain errors.

One of the early but still popular codes for this purpose is the Hamming code. The Hamming code uses overlapping parity bits to pinpoint the location of any single error in a message.

One common instance of the Hamming code is H(7,4) which encodes four bits of data in seven total bits (three bits used for error correction). This code can correct any single bit error.

In this Hamming code, the three parity bits are located at the first, second, and fourth positions. The first par-



ity bit is defined by $p_1 = d_1 \oplus d_2 \oplus d_4$. The second parity bit is defined by $p_2 = d_1 \oplus d_3 \oplus d_4$. The final parity bit is defined by $p_3 = d_2 \oplus d_3 \oplus d_4$.

The receiver recalculates the parity to check if an error exists. If all the received parity bits match the calculated parity bits, no error exists. Otherwise, to find the position of the error, start with 0. If $r_1 \neq p_1$, add 1. If $r_2 \neq p_2$, add 2. If $r_3 \neq p_3$, add 4. This value will show where in the *encoded* message the error is found.

■ **Example 15.3** • For example, assume the message 0101_2 is to be transmitted using the H(7,4) code. In this message, d_1 through d_4 correspond to the four bits of the message; with $d_1 = 0$, $d_2 = 1$, $d_3 = 0$, and $d_4 = 1$. The parity bits will be computed as shown, namely $p_1 = d_1 \oplus d_2 \oplus d_4 = 0 \oplus 1 \oplus 1 = 0$, $p_2 = d_1 \oplus d_3 \oplus d_4 = 0 \oplus 0 \oplus 1 = 1$, and $p_3 = d_2 \oplus d_3 \oplus d_4 = 1 \oplus 0 \oplus 1 = 0$.

The final message is constructed following the pattern $p_1p_2d_1p_3d_2d_3d_4$. The encoded message is thus 0100101_2 .



■ **Example 15.4** • Imagine this message is sent over a noisy channel, and an error occurs. The receiver receives $010010\textcolor{orange}{0}_2$. To check the value, the receiver first separates the parity bits from the data, as shown above. The received version has parity $r_1 = 0$, $r_2 = 1$, $r_3 = 0$ and data $010\textcolor{orange}{0}$. The receiver still does not know that an error exists in the data.

The receiver now recalculates the expect parity using the technique above. They find the expected parity is $p_1 = 1$, $p_2 = 0$, $p_3 = 1$. These parity values do not match the received parity values, thus, an error exists. To pinpoint the error, take each calculated parity bit which does not match the received parity bit (in this case, all of them) and sum the positions (with the leftmost position being 1) that those parity bits appear in the message.

In this case, since all three parity bits do not match the expected values, the error is at position $1 + 2 + 4 = 7$. Counting the leftmost bit as number 1, bit number 7 pinpoints the error.

The seemingly random mix of parity with data allows a fast mathematical technique known as matrix multiplication to be used to encode messages.

$$\sum_{n=1}^{\infty} \frac{1}{n^s}$$



An error which damages only the parity and not the data is still considered an error because the error correction process must be undertaken before the receiver will know whether or not a data error has occurred.



The error can be corrected by flipping the offending bit. ■

■ **Example 15.5** • An error can occur in parity as well. Imagine the previously encoded message, 0100101_2 , was received with one error: $010\textcolor{orange}{1}101_2$.

In this case, the received parity is $r_1 = 0, r_2 = 1, r_3 = 1$. The receiver again starts by calculating the expected parity. The expected parity is $p_1 = 0, p_2 = 1, p_3 = 0$. The expected and received parities do not match, so an error has been found.

The location of the error is pinpointed by noting which parity bits don't match (just $r_3 \neq p_3$ in this case), and adding the position value for the non-matching bits. In this case, the error is found in position 4 in the encoded message. Position 4 is the third parity bit, indicating that the error has not damaged the data. ■

An alternative approach to finding the location of the error is to view each parity bit as representing a set of data bits that it is based on. So $P_1 = \{d_1, d_2, d_4\}$, $P_2 = \{d_1, d_3, d_4\}$ and $P_3 = \{d_2, d_3, d_4\}$. To find the location of an error, take the intersection of all the sets, with the complement of those sets whose parity bit (calculated vs received) matches.

In the above examples, we first saw the case when all three parity bits (calculated vs received) were mismatched. In that case, the error was in $P_1 \cap P_2 \cap P_3 = \{d_4\}$. In the case where a parity bit itself is in error, the result of the intersection will be the empty set, since no data bit is in error. As above, $P'_1 \cap P'_2 \cap P_3 = \emptyset$.

15.5 Reed-Muller Code

The Reed-Muller code, used in applications such as early space probe imaging transmissions, is able to correct more errors although at a cost of a lower code rate. A particular instance of the Reed-Muller code is the Hadamard code which provides a simple construction and excellent error correcting properties.



The Hadamard code is constructed by starting with a small 2-D array:

$$\begin{array}{cc} 1 & 1 \\ 1 & 0 \end{array}$$

The seed must be grown to construct an array. Each growth increases the number of data bits by one and also increases the number of correctable errors by one. A technique to grow a Hadamard array is to make three copies of it and arrange them with the original in a square, with the bottom-right copy inverted. Finally, after the seed has been grown, a copy is made, inverted, and placed below. This procedure sounds complicated but can be quickly realized through the use of examples.

Taking the seed above, we will perform no growth steps and go immediately to the final copy-and-invert. The leftmost column indicates the binary value of the data, and the row across indicates the encoded message to transmit.

00	1	1
01	1	0
10	0	0
11	0	1

The above array is RM(2,2) which encodes two bits of data into two total bits, and has no capability for error detection. The value to be transmitted is found by row (the first row corresponds to 00, the second to 01, and so on), and then the entire row is transmitted.

As the array expands, the error correction capabilities improve. Performing a single growth step before the copy-and-invert yields RM(4,3).



000	1	1	1	1
001	1	0	1	0
010	1	1	0	0
011	1	0	0	1
100	0	0	0	0
101	0	1	0	1
110	0	0	1	1
111	0	1	1	0

In addition to the square, copy and invert technique, it is possible to construct the next Hadamard array by taking any existing Hadamard array and replacing each 1 with a copy of the 2×2 seed, and each 0 with a copy of the inverse of the 2×2 seed.



The above array is RM(4,3) which encodes three bits of data into four total bits, and can detect (but not correct) any one error. In this array, the data value 0 would be transmitted as 1111_2 , while the data value 6 would be transmitted as 0011_2 . Recall that the value to transmit indicates which row to select; the entire row is then transmitted.

0000	1	1	1	1	1	1	1	1
0001	1	0	1	0	1	0	1	0
0010	1	1	0	0	1	1	0	0
0011	1	0	0	1	1	0	0	1
0100	1	1	1	1	0	0	0	0
0101	1	0	1	0	0	1	0	1
0110	1	1	0	0	0	0	1	1
0111	1	0	0	1	0	1	1	0
1000	0	0	0	0	0	0	0	0
1001	0	1	0	1	0	1	0	1
1010	0	0	1	1	0	0	1	1
1011	0	1	1	0	0	1	1	0
1100	0	0	0	0	1	1	1	1
1101	0	1	0	1	1	0	1	0
1110	0	0	1	1	1	1	0	0
1111	0	1	1	0	1	0	0	1

The above array is RM(8,4) which encodes four bits of data into eight total bits, and can correct one error. At this point, the Hamming code is still superior, encoding more data bits into less total bits and correcting one error. However, as the array expands, more error correcting is possible.

The next array, RM(16,5), can correct up to three errors. The subsequent array, RM(32,6), can correct up to seven errors.



The error correction capabilities of this code derive from the Hamming distance between each row in the array. When a row is received, the receiver finds which row in the array is most similar (has the lowest Hamming distance) to the received row. As the array grows, the minimum Hamming distance between the rows also grows.

The general properties of the Hadamard Reed-Muller code specify that n data bits will be represented with 2^{n-1} total bits, and be able to correct up to $2^{n-3} - 1$ errors. One additional error can be detected, but not corrected. If additional errors have occurred, then the received error will be closer (in Hamming distance) to an different original row, causing a wrong value to be selected.

■ **Example 15.6** • For example, using the RM(8,4) array shown above, imagine the sender wants to transmit the value 0110_2 . This corresponds to the row $1100\ 0011_2$. This eight bit sequence is transmitted to the receiver, who receives $1100\ 1011_2$, with a single error. The receiver notes that this eight bit sequence does not directly correspond to any of the eight bit sequences shown in the RM(8,4) array. The receiver then determines the Hamming distance (how similar/different) the received row is to each other possible row. The table below highlights differences. The Hamming distance (number of differences) is shown in the right-most column.

Any code whose possible values have a minimum Hamming distance of d will be able to correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors.



0000	1	1	1	1	1	1	1	1	3
0001	1	0	1	0	1	0	1	0	3
0010	1	1	0	0	1	1	0	0	3
0011	1	0	0	1	1	0	0	1	3
0100	1	1	1	1	0	0	0	0	5
0101	1	0	1	0	0	1	0	1	5
0110	1	1	0	0	0	0	1	1	1
0111	1	0	0	1	0	1	1	0	5
1000	0	0	0	0	0	0	0	0	5
1001	0	1	0	1	0	1	0	1	5
1010	0	0	1	1	0	0	1	1	5
1011	0	1	1	0	0	1	1	0	5
1100	0	0	0	0	1	1	1	1	3
1101	0	1	0	1	1	0	1	0	3
1110	0	0	1	1	1	1	0	0	7
1111	0	1	1	0	1	0	0	1	3

The entry with the lowest Hamming distance between itself and the received sequence is the row for 0110_2 , so the receiver will assume that is the desired value. ■



15.6 Exercises

Solutions to these exercises can be found in Appendix A.15 on page 327.

1. *Problem:* Find the Hamming Distance between the two messages $0101\ 1001_2$ and $0111\ 1010_2$.
2. *Problem:* The following messages have an even parity bit appended to the end and were transmitted over a possibly noisy channel, with at most one error. Determine which messages contain an error.
 - (a) $1111\ 1111_2$
 - (b) $1010\ 1011_2$
 - (c) $1111\ 0000_2$
 - (d) $0000\ 0000_2$
3. *Problem:* Construct a logic circuit which accepts 4 data bits as input and indicates the correct even parity bit for these four data bits.
4. *Problem:* The following messages, each with 8 data bits, have been encoded using the Berger code and transmitted over a channel which may only convert 1s to 0s. Determine which messages contain error(s).
 - (a) $0101\ 1001\ 0100_2$
 - (b) $1101\ 0011\ 0001_2$
 - (c) $0011\ 0111\ 0010_2$
 - (d) $1110\ 0111\ 0010_2$
5. *Problem:* Encode the following messages using the H(7,4) Hamming code.
 - (a) 0001_2
 - (b) 0000_2
6. *Problem:* The following messages have been encoded with the H(7,4) Hamming code and transmitted over a possibly noisy channel, with at most one error. Determine which messages contain an error. In all cases, find the original 4-bit message.
 - (c) 1110_2
 - (d) 1010_2
7. *Problem:* Construct the array for the Hadamard code RM(16,5).
8. *Problem:* The following messages have been encoded using the RM(16,5) Hadamard code and transmitted over a possibly noisy channel, with up to four errors. Determine which messages contain error(s) and how many error(s) are present. In cases with up to three errors, find the original 4-bit message. In cases with four errors, show why recovering the original 5-bit message is not possible.
 - (a) $1010\ 1010\ 1010\ 1010_2$
 - (b) $0111\ 1000\ 0000\ 0111_2$
 - (c) $0011\ 1110\ 1100\ 0011_2$
 - (d) $0011\ 1100\ 0011\ 0000_2$
 - (e) $1000\ 1110\ 1101\ 0100_2$

Chapter 16

Compression

PART V

Algorithms

17 Digital Logic	170		
17.1 Addition	170	19.7 Functions and Reuse	215
17.2 Subtraction	178	19.8 Logical Expressions and Set Extraction	217
17.3 Comparison	183	19.9 Exercises	222
17.4 Encoders and Decoders	186		
17.5 Feedback Circuits	188	20 Analysis of Algorithms	223
17.6 Latches and Flip-Flops	190	20.1 Performance Factors	224
17.7 Exercises	195	20.2 Complexity Classes	225
18 Finite Automata	196	20.3 Binary Search: A Practical Example	226
19 Flowcharts	197	20.4 Step Counting	229
19.1 Introduction to Algorithms	197	20.5 Loop Analysis	230
19.2 Fundamental Control Structures	199	20.6 Comparison of Complexity Classes	232
19.3 Selection Control Structures	202	20.7 Exercises	236
19.4 Repetition Control Structures	205		
19.5 Other Symbols	208	21 Expression Trees	238
19.6 Managing Lists	211		

Chapter 17

Digital Logic

 Due to universality, it would be possible to construct an entire computer processor out of only NAND gates, for instance.

Previously, we indicated that logic gates are the fundamental constructs of physical computation. Using logic circuits, operations such as addition and subtraction, memory, decisions, and counting become possible using nothing more than basic and compound logic gates.

The central processor and other supporting components inside of a computer consist of many of these constructs interconnected to allow a variety of fundamental data operations to be performed. In these examples, for simplicity, we will show small bit widths (usually 4 or 8 bits) and largely isolated circuits. Modern processors are usually constructed to perform operations at 64 bits, and most have a substantial number of hardware implemented instructions (more than 50), various levels of memory and cache, and other optimizations. The study of CPU design is an interesting field, but far beyond the scope of this text.

17.1 Addition

We have seen that arbitrary Boolean expressions can be translated into logic circuits. However, what expression (and, correspondingly, logic circuit) could represent addition? First, recall the fundamental two bit binary addition:

$$\begin{array}{rcl}
 0 & + & 0 = 0 \\
 1 & + & 0 = 1 \\
 0 & + & 1 = 1 \\
 1 & + & 1 = 10
 \end{array}$$

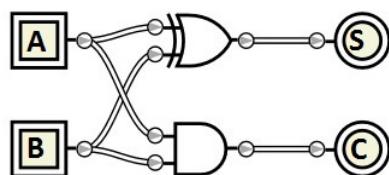
There are two inputs, and two outputs. The reason two outputs are needed is that $1 + 1 = 10$, which has two bits. In wider additions, the leftmost bit of a two-bit result is carried, so we will call this bit the carry. The rightmost bit is the sum. We can rearrange this result to follow the pattern of a truth table, with 1 becoming True, and 0 becoming False.

a	b	Carry	Sum
F	F	F	F
T	F	F	T
F	T	F	T
T	T	T	F

It would be possible to generate a logical expression for each output using disjunctive normal form, but perhaps a shorter expression could be eyeballed in this case. Looking at the output Carry, what operator does this seem to be based on? It is true when both inputs are true, and only then. Thus, Carry can be represented by AND. Likewise for Sum, it is true when either but not both of its inputs are true. Thus, Sum can be represented by XOR. If a and b are inputs, then the logical expressions for these outputs could be written as $s = a \oplus b$ and $c = a \wedge b$.

These expressions can now be formed in a logic circuit, known as a half-adder.

 *Half Adder:* logic circuit that can add two bits.



However, a half adder is of minimal practical use. In order to add a number that is more than one bit in size,

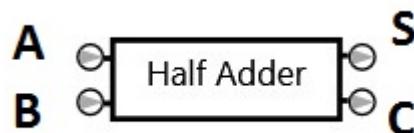
the carry result must be dealt with from one column to the next. This means that every addition column (except the right-most one) for two binary numbers actually has three inputs: the respective bits from the numbers, as well as the carry from the prior column.

Recall the table for three bit addition:

0	+	0	+	0	=	0
0	+	1	+	0	=	1
0	+	0	+	1	=	1
0	+	1	+	1	=	10
1	+	0	+	0	=	1
1	+	1	+	0	=	10
1	+	0	+	1	=	10
1	+	1	+	1	=	11

There are still only two outputs, the sum bit and the carry bit to the next column. We could follow a similar process as previously, constructing a truth table and then deriving an expression. However, given that we have already defined two bit addition, we can take advantage of the associative property of addition to note that $a + b + c = (a + b) + c$. That is, the half adder can add the first two bits, and the result can be added to the third bit with a second half adder.

It is very common to duplicate portions of circuits to serve larger inputs; in order to avoid manually duplicating gates which could lead to a large and confusing circuit, we will conceptually package a block of gates to reuse into a named box. Inside the box are the exact same gates that have already been developed. For instance, here is a package for half adder:

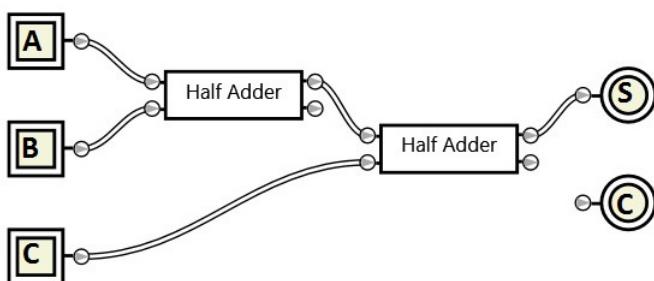


This half adder package can now be used as if it were a single gate in its own right, with inputs and outputs,



but every one of these will actually represent an AND and an XOR, as shown previously.

Performing three bit addition using two half adders may seem straightforward in terms of calculating the sum, but each half adder has its own carry output, while the overall circuit itself has only one carry output. It is not immediately clear how to connect the carry outputs.



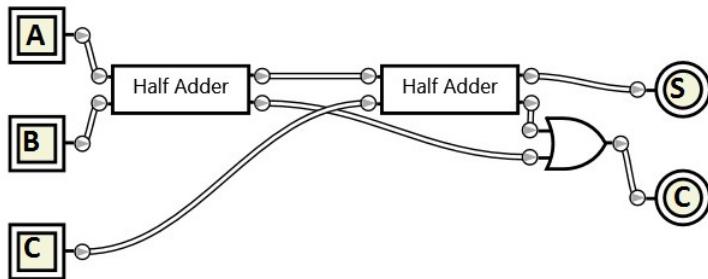
To determine how the carry works, consider the original three bit addition table. In this case, the carry out is true only when two or more inputs are true. Note that in the half adder, the carry output is implemented as AND, which will show if two inputs are true. Thus, if either half adder has a true carry, there are two true values being input into that half adder, and so the final addition will have a true carry out.

The completed circuit is known as a full adder.

Either an OR or an XOR may be used to combine the carry outs, as it is not possible that both half adders will produce a true in their carry outputs at the same time.



Full Adder: logic circuit that can add three bits.



The ability to add three bits corresponds to the two inputs and carry in that each column of binary addition performs. Thus, to add any two binary numbers both of n bits, we need n full adders connected together. For



Once larger adders are constructed in this way, they themselves can be chained together. For example, four 4-bit adders could be connected to make a 16-bit adder.



each full adder, the sum output goes to the corresponding column in the addition result, while the carry goes to the next column's full adder. To understand how this procedure works, perform a binary addition by hand and note that each column produces one bit of the sum and possibly a carry bit to the next column.

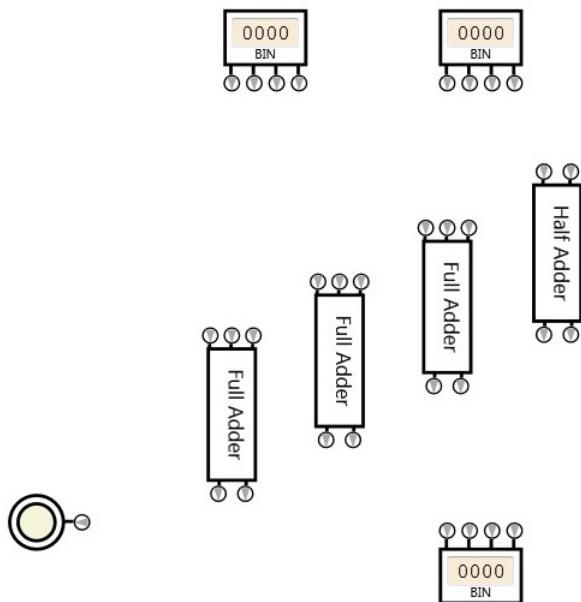
In order to add two numbers of n bits, we will need multiple full adders (one for each bit). Thus, we will create a package for full adder so that the structure of a full adder (which consists of nothing more than AND, OR, and XOR) can be duplicated easily.

We can simplify (for the moment) this mechanism slightly by making the first (rightmost) adder a half-adder, since there will never be any carry in to the rightmost addition.

Besides the four sum bits, the final leftmost full adder has a carry out that is unaccounted for. We could extend the size of output to five bits, however, if we are considering the techniques using in computer design, the number of bits is largely fixed because the output from one operation usually proceeds into another operation. If we increase the output size to five bits, then the input sizes would need to be increased, which would add an extra full adder, increasing the output size to six bits and so on, forever.

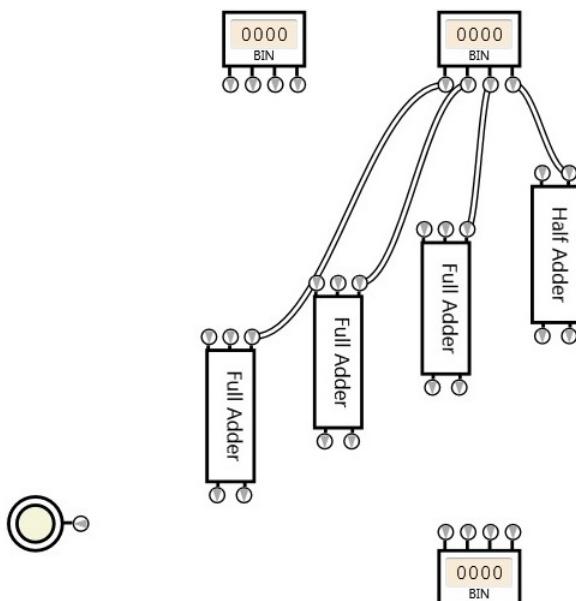
Thus, the last carry out is not part of the output proper, but instead indicates (for unsigned integers) when the sum of the two n bit numbers cannot be represented in n bits; this is the overflow condition.

■ **Example 17.1** • For example, we will consider a four bit addition circuit. The wiring is somewhat complex, so it serves to show each segment piecemeal first, and then connect them all together.

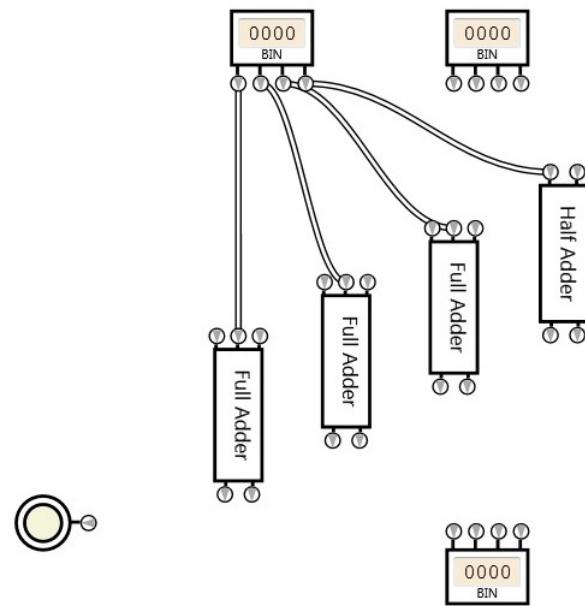


First, we show the wiring for one of the 4-bit input blocks. Each bit in the input connects to the corresponding input bit on each adder.

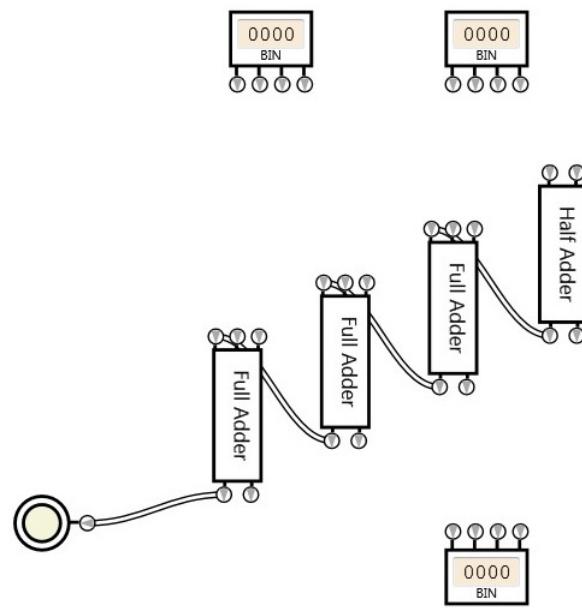
Due to the commutative property of addition, it does not matter which input on the adder a particular bit is connected to. However, the columns must be correct. So the rightmost bit must connect to the rightmost adder, but it does not matter which input on the rightmost adder is selected, and so on.



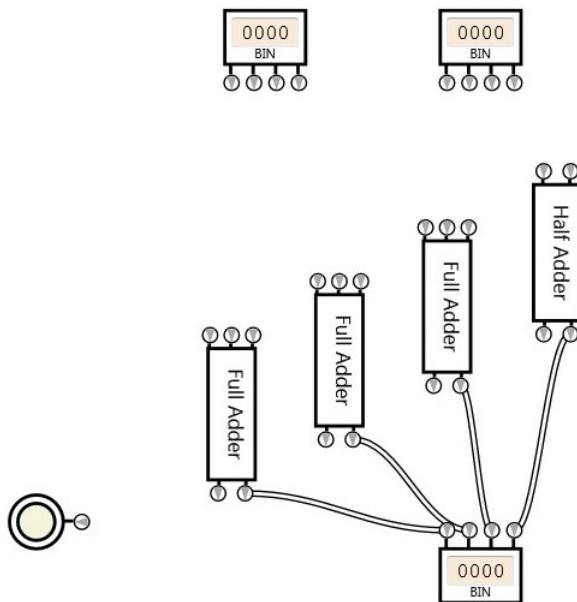
Next, we show the same wiring for the other 4-bit input block.



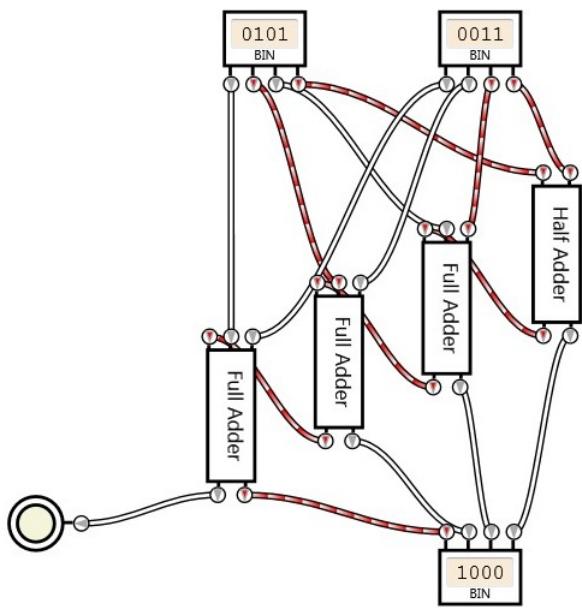
When the carry out wires are connected, each adder's carry out connects to the input of the next adder. For the last adder, the carry out connects to an overflow indicator. When the indicator activates, the two inputs (if unsigned integers) cannot be successfully represented in 4 bits.



The sum outputs from each adder connect to the corresponding sum columns in the result.



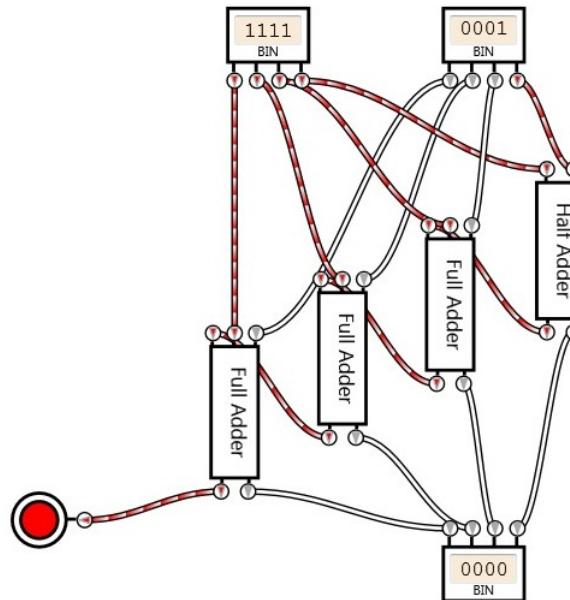
Here is the entire four bit adder, in operation, showing that $0101_2 + 0011_2 = 1000_2$.



17.2 Subtraction

In a previous chapter, the value of Two's Complement as a representation of negative was affirmed in particular because it allowed the same circuit to perform both addition and subtraction. We will use the 4 bit adder created in the previous section with only a few minor changes to perform addition and subtraction.

First, when switching mentally from unsigned to signed representation, the overflow indicator ceases to be meaningful. If there is any doubt about this, attempt $-1_{10} + 1_{10} = 0_{10}$ in four bit Two's Complement, using the adder shown previously: $1111_{2C} + 0001_{2C} = 0000_{2C}$, which is the correct result, but the overflow indicator turns on, although this indicates no error condition when dealing with signed numbers.



Without making any changes, the same addition circuit can now be used for subtraction by transforming the request $a - b$ into $a + (-b)$ and finding the Two's Complement representation for $-b$ as appropriate.

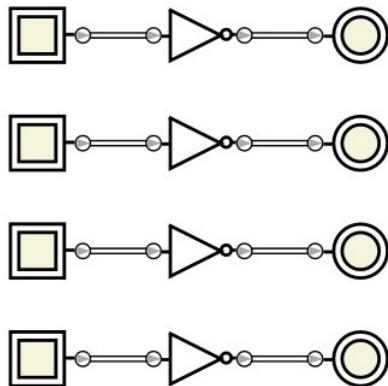
However, it would be nice if the circuit could perform the Two's Complement work for us. So, for instance, we could enter two positive values a and b and indicate



if we want $a + b$ or $a - b$ to be computed. If $a - b$ is selected, then the Two's Complement of $-b$ must be found.

The mechanical technique for finding the inverse (negative) of any Two's Complement value is to invert and add one. We will consider these steps separately. First, to invert. If we always wanted to invert the four bit value, this could be accomplished using NOT gates.

$\sum_{m=1}^{\infty} \frac{1}{n^m}$ An inverter applies the bit-wise NOT operator, \sim



However, the main weakness of this approach is that there is no capability to select whether or not the inverter should be active. Thus, given two positive values a and b , we would only be able to perform $a - b$. If we want to be able to select between addition and subtraction, the inverter must have an additional input. If the input is true, then the four bits will be inverted. If the input is false, then the four bits will not be inverted.

How can such a selectable inverter be constructed? First, consider the simplest version: a single bit selectable inverter. This circuit would have two inputs (one for data, and one for the invert selection) and one output (the data, either inverted or not). To determine how this circuit should be implemented, first design a truth table for it. Let d indicate the input data, and i indicate whether or not to invert.

d	i	Output
T	T	
T	F	
F	T	
F	F	



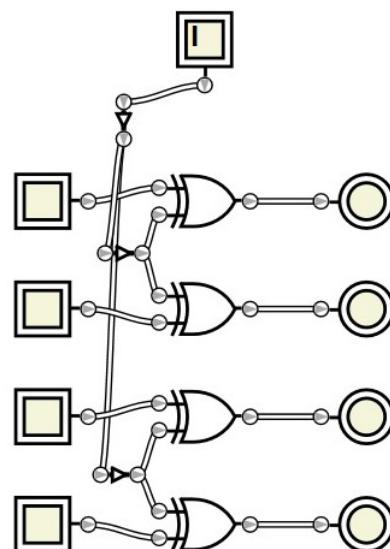
First, if the invert flag i is set to false, then the data d should be passed directly to the output.

d	i	Output
T	T	
T	F	T
F	T	
F	F	F

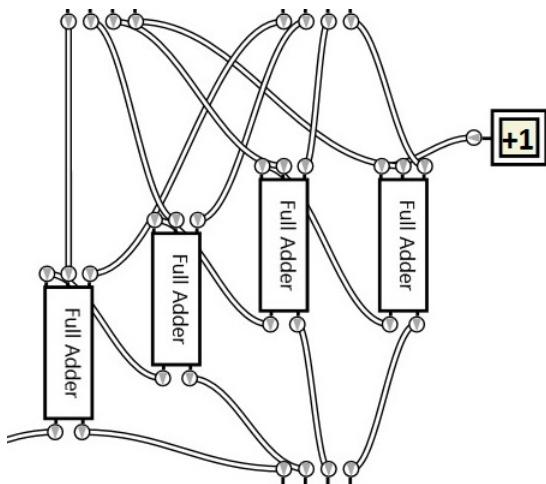
Next, if the invert flag i is set to true, then the output should be the opposite of the data d .

d	i	Output
T	T	F
T	F	T
F	T	T
F	F	F

What expression meets the criteria shown by this truth table? This truth table can be represented in one operation using XOR. For every bit that we may want to invert, we will connect that data bit to an XOR gate, and the invert flag will connect to all XOR gates. Thus if the invert input is set to false, the data will pass through normally; if it is set to true, the data will be inverted.

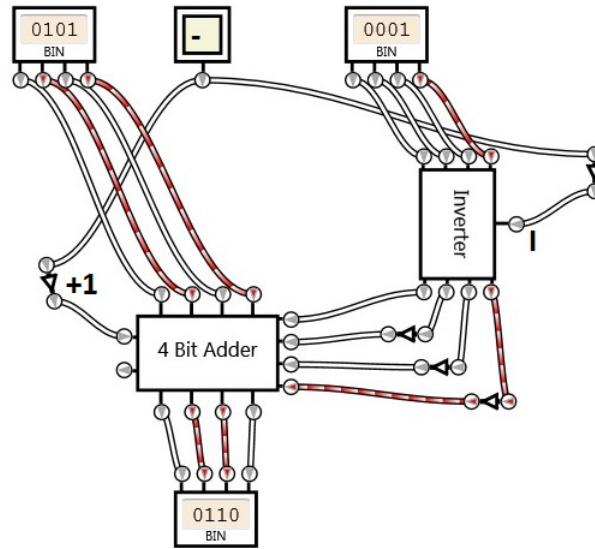


This inverter can be used to solve the invert problem. But how can the add one be applied on request as well? Look back to the original four bit adder. The rightmost adder was a half adder instead of a full adder, because there was no possibility for a carry in on the first bit. However, if a full adder were used, this carry in, if set, would have the effect of adding one to the result.



The final circuit includes the inverter and four bit adder, extended to accept a carry in. An additional input indicating whether or not subtraction should be performed is included.

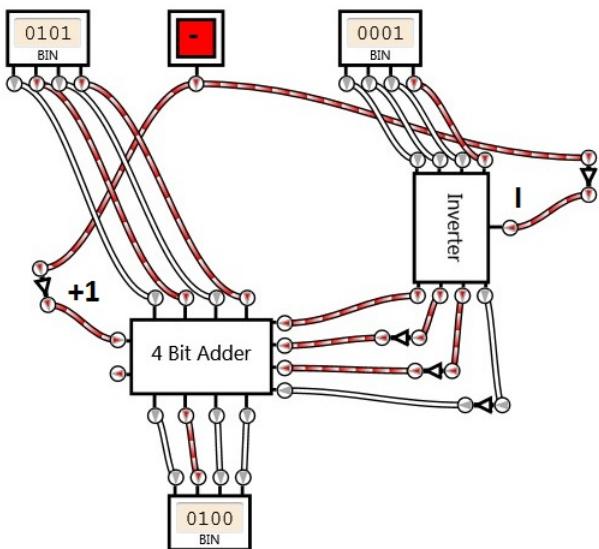
- **Example 17.2** • For example, here the circuit is demonstrated adding $5_{10} + 1_{10} = 6_{10}$. Each is first represented as Two's Complement positive numbers, giving the inputs $0101_{2C} + 0001_{2C} = 0110_{2C}$. Note that because the subtraction mode is not selected, the inverter does nothing, so the input passes through it harmlessly. Likewise, there is no additional input into the adder, so the adder considers only the exact values given here, producing a straightforward binary addition.



■ **Example 17.3** • If the circuit is switched into subtract mode without changing the inputs, then it will compute $5_{10} - 1_{10} = 4_{10}$. The steps involved here are slightly more complex. The subtract mode, together with the addition circuit, means that the compute will actually, in some sense, process $5_{10} + (-1)_{10}$. First, the subtract mode activates the inverter, which takes the input 0001_2 and turns it into 1110_2 . The correct representative for -1_{10} is 1111_{2C} , so we can see that the inverted value is short by one. This makes sense as the mechanical conversion procedure is to invert AND add one.

The subtract flag also goes to the carry in on the full adder, which adds one to sum, taking care of the add one portion of the conversion. Thus, the system mechanically transforms the expression as follows $0101_{2C} - 0001_{2C} = 0101_2 + (-0001_2) = 0101_2 + (\sim 0001_2) + 1_2 = 0101_2 + 1110_2 + 1_2 = 10100_2$. Keep in mind that our output is only four bits, so the extra bit triggers the overflow (which is now meaningless in Two's Complement mode) and our final result is 0100_{2C} , which is the correct subtraction results.





■

This circuit will work for all Two's Complement values, positive and negative, and will produce a correct result as long as the result can be represented with four bit Two's Complement. When dealing with Two's Complement, the overflow output is not sufficient to determine if the result could not be represented. A more sophisticated error detection mechanism will be needed.

17.3 Comparison

Another common task for computers is to compare two values to determine which one is smaller, or if the two values are equal. Like addition, a multi-bit comparison circuit can be constructed incrementally by first considering the most basic comparison and then building them together.

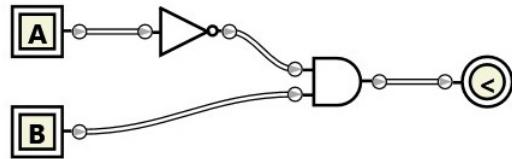
Imagine comparing two bits, a and b . If we want to determine whether $a < b$, $a \leftrightarrow b$, or $a > b$, this at first looks like three operations. However, it is in fact only two: if we get a false result on any two operations, then the third must be true. For our purposes, we will implement $a < b$ and $a \leftrightarrow b$, and then let $a > b$ be defined as $\neg((a < b) \vee (a \leftrightarrow b))$.



The equality operator, \leftrightarrow , is already known to be implemented with XNOR. However, how can the operator $<$ be implemented? We can begin by constructing a truth table. Recall that T maps to 1 and F maps to 0, therefore we will use 0 and 1 instead of F and T in the inputs of the this truth table, as numeric comparison is our goal.

a	b	$a < b$
1	1	F
1	0	F
0	1	T
0	0	F

Thus, $a < b$ can be represented with the expression $\neg a \wedge b$.

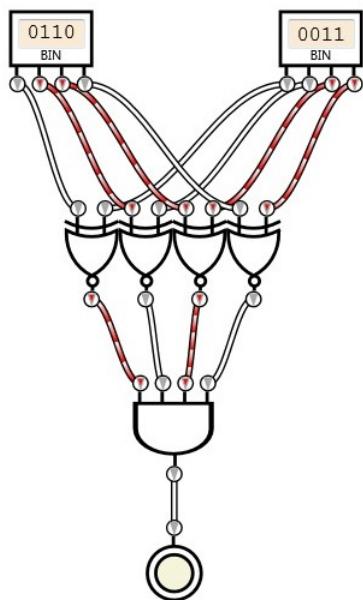


Next, comparison must be extended to multiple bits. For equality, each respective bit pair will be sent into an XNOR gate. This gate will output true if the bit pair is equal, and false if not. Equality is defined as all bits being equal, so the results of these XNOR gates will then be placed into an AND, so that the two input numbers are considered equal only if all their bit pairs are equal.

■ **Example 17.4** • Four bit equality can be defined first with two four-bit inputs a and b , broken down into bits: $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$. Each respective position will paired with XNOR, and combined with AND: $(a_3 \leftrightarrow b_3) \wedge (a_2 \leftrightarrow b_2) \wedge (a_1 \leftrightarrow b_1) \wedge (a_0 \leftrightarrow b_0)$.

A logic gate implementation, shown here, demonstrates how 0110_2 and 0011_2 are not equal due to their difference in two positions.





■

To process $a < b$ with multiple bits, the general procedure for comparing binary values must be devised. Given two binary numbers of equal length, to find if $a < b$, follow these steps (note that this is fundamentally the same way you would determine which of any two numbers of any base are less):

1. Start with the left-most bit of both numbers.
2. If the bit from a is 0, and the bit from b is 1, then $a < b$.
3. If the bit from a is 1, and the bit from b is 0, then $a > b$.
4. Otherwise, the bits are equal. If bits remain, move one bit to the right and go to step 2.
5. If no bits remain, the numbers are equal.

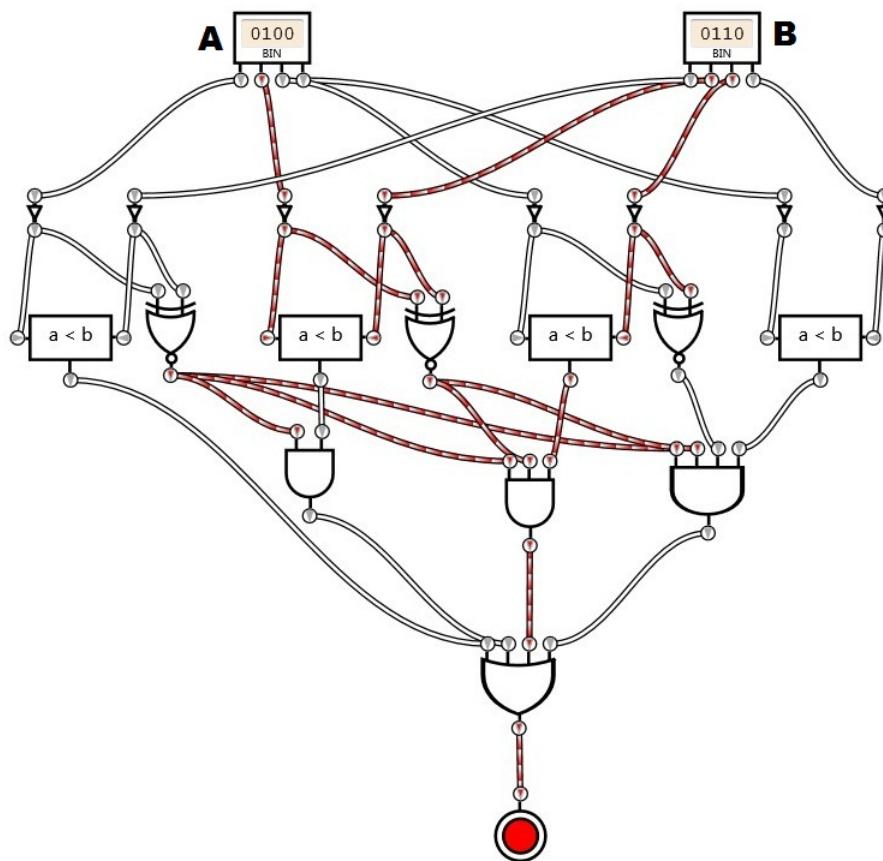
■ **Example 17.5** • Four bit less-than can be defined incrementally by first defining the comparison step for each bit, and then chaining them in the appropriate order. As before, define two four-bit inputs a and b , broken down into bits: $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$.



Using the definition of $<$ described above, namely that for any pair of bits, $x < y$ is defined as $\neg x \wedge y$, we can define $a < b$ as $(a_3 < b_3) \vee ((a_3 \leftrightarrow b_3) \wedge (a_2 < b_2)) \vee ((a_3 \leftrightarrow b_3) \wedge (a_2 \leftrightarrow b_2) \wedge (a_1 < b_1)) \vee ((a_3 \leftrightarrow b_3) \wedge (a_2 \leftrightarrow b_2) \wedge (a_1 \leftrightarrow b_1) \wedge (a_0 < b_0))$.

Although this expression looks complex, it follows from the step-by-step implementation given above, where each bit is compared from the right, and only if all bits so far are equal are further comparisons made.

A logic gate implementation shown here, demonstrates how $0100_2 < 0110_2$.



■

17.4 Encoders and Decoders

Many essential components of a computer require indicating which task, operation, or setting to select by use of a binary number. For this to be accomplished,



a given number must be split into each possible value. An n bit binary number has 2^n possible values. Establishing the relationship between these are the responsibility of encoders (which have 2^n inputs and n outputs) and decoders (which have n inputs and 2^n outputs).

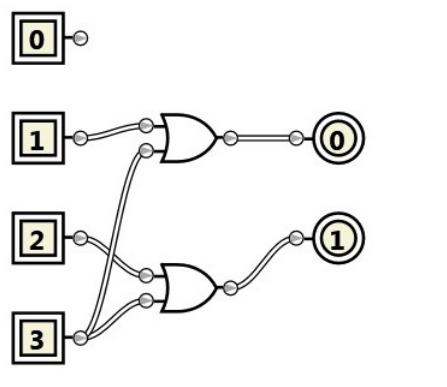
In a simple encoder, exactly one of the 2^n input lines is expected to be true. The encoder will then output the binary value associated with the particular, selected, input line.

■ **Example 17.6** • The abbreviated truth table below shows a 4 to 2 encoder. Notice that the value of the output O is equal to the binary value of the particular selected input line I .

I_3	I_2	I_1	I_0	O_1	O_0
F	F	F	T	F	F
F	F	T	F	F	T
F	T	F	F	T	F
T	F	F	F	T	T

An astute reader will note that a truth table with four inputs should have 16 rows, but this only shows four. In a simple encoder, the output values are only defined when exactly one input is true. To handle the case when multiple inputs are true, a priority encoder can be used.

A logic gate implementation of a 4 to 2 simple encoder is shown here.



Encoder: logic circuit that converts from one input line into an equivalent binary number.



Priority Encoder

Priority Encoder: an encoder which outputs the binary value equivalent to the highest active input line.



The I_0 line is not connected because if the input 0 is selected, then no outputs will be on, as the value 0 is represented by all 0s in binary.





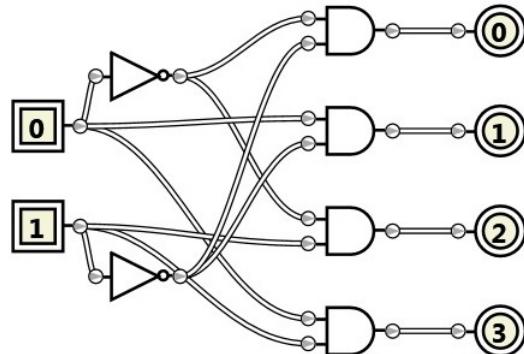
 **Decoder:** logic circuit that converts from input into one output representing the input's binary value.

A decoder performs the opposite task of an encoder. Given any combination of input values, it will provide exactly one true output indicating which number is formed from the inputs. Decoders are useful for addressing memory and selecting CPU instructions, among other tasks.

■ **Example 17.7** • The truth table for the 2 to 4 decoder is shown below.

I_1	I_0	O_3	O_2	O_1	O_0
F	F	F	F	F	T
F	T	F	F	T	F
T	F	F	T	F	F
T	T	T	F	F	F

A straightforward logic gate implementation uses disjunctive normal form for each output.



17.5 Feedback Circuits

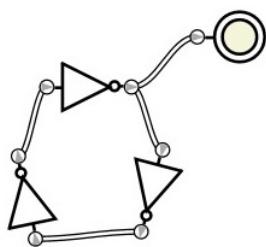
In previous chapters, we have seen that Boolean expressions, truth tables, and logic circuits are all interconvertible. This is generally true, however, only with



respect to feed-forward logic circuits. It is also possible to create a logic circuit where the output from one portion of the circuit feeds back, as a loop, into earlier portions of the circuit. Such circuits do not have a strict fixed input gives fixed output behavior, but may change their output based on previous values, or even change output without change of input.

Such circuits are essential to computers, as they allow for memory, timing, and dynamic behavior. In many cases, signal propagation speed, a physical attribute of the implementation, becomes an important factor in circuit design.

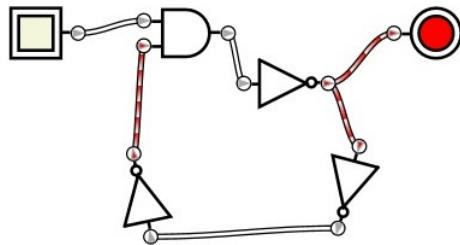
■ **Example 17.8** • The simplest, although not useful, feedback circuit is a cycle of three NOT gates. The input to the first NOT gate will, mathematically be the same as its output, which will cause the first NOT gate to invert, and the process will repeat. This circuit will alternate its output between true and false as rapidly as the signal can propagate.



In certain feedback circuits, a stateful truth table can be used which takes into account the previous output from the circuit in defining the behavior or new output. There may also be undefined states where certain combinations of inputs create unstable behavior.

■ **Example 17.9** • A modified version of the three NOT cycle includes a toggle input. When the input is set to false, the circuit is in a stable state. When the input is set to true, the output of the circuit alternates.





A stateful truth table for this circuit considers that, in alternation, the output is the opposite of the previous output. Let I be the input and O be the output.

I	O_{prev}	O
T	T	F
T	F	T
F	T	F
F	F	F

■

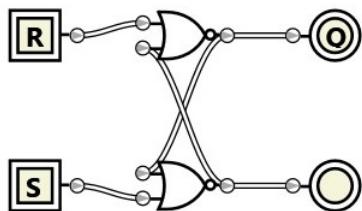
17.6 Latches and Flip-Flops

A latch is a single bit memory circuit which retains a value. The value can be changed at any time as soon as the command is received. The circuit will retain the current value until a change command is given.

■ **Example 17.10** • One of the simplest latch circuits is the SR latch, which is short for Set-Reset Latch. This latch has two inputs, one for the set command, and one for the reset command. The SR latch is composed of two NOR gates, with the output from each going to the other as one of the inputs. The circuit has an output, and its complement.

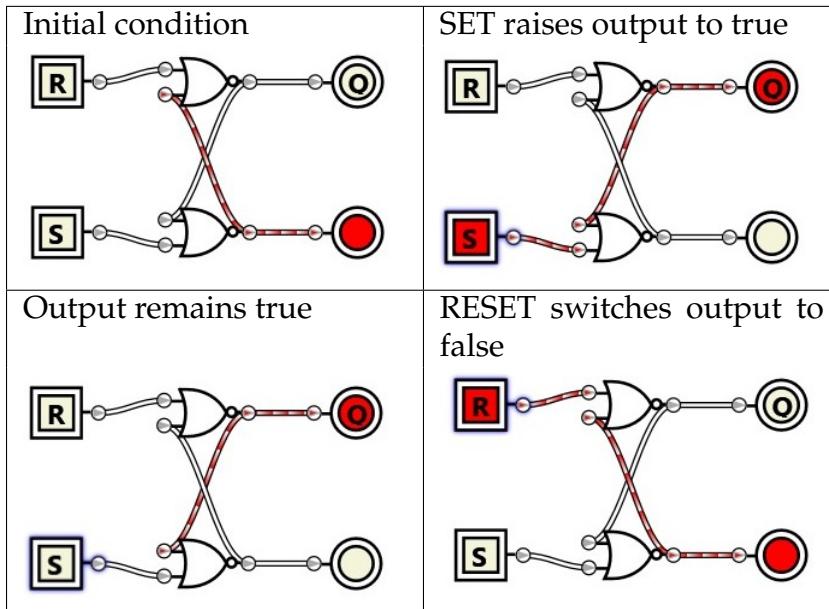


SR Latch: simple one bit memory cell which set either be Set (true) or Reset (false).



When both inputs are false, the current state is maintained. When the set input goes to true, the output becomes true. When the reset input goes to true, the output becomes false. Note that only at most one of these inputs may be true at any given time; if both set and reset are true, the behavior is formally undefined.

 SR Latches can also be constructed with NAND gates. This implementation may be more desirable for cost or complexity reasons.



A stateful truth table for the SR latch can be given. Note that if both S and R are true, the behavior is undefined.

S	R	Q_{prev}	Q
T	T	T	undefined
T	T	F	undefined
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	T
F	F	F	F



In many cases with complex logic circuits, an abbreviated truth table using the decision table rules of indifferent conditions, may be more informative. Here is the table above, simplified.

S	R	Q_{prev}	Q
T	T	-	undefined
T	F	-	T
F	T	-	F
F	F	T	T
F	F	F	F



 **Clock:** a regular pulse signal which synchronizes the activities of logic circuits throughout a component.

 The clock input is often indicated in a logic circuit with the $>$ symbol.

 **D Latch:** one bit memory cell which holds the given data when the clock input is true.

As circuits become more complex, the timing of interactions becomes increasingly significant. Delays in signal propagation can result in fluctuation of values before the final, correct value is settled. To ensure that memory selects the correct value, a clock is often used. A desirable memory cell will only save a value at the precise moment indicated by a clock pulse, and ignore all other variations in input.

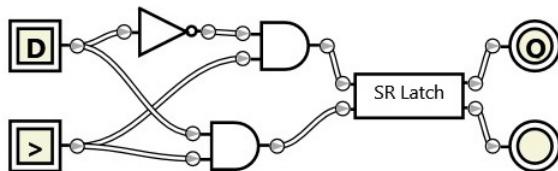
The time required for signal propagation through various components limits the maximum speed of the clock. Computer manufacturers advertise clock speed as one aspect that can effect overall computational performance.

■ **Example 17.11** • The first improvement is to only accept input when a clock signal is present. The D Latch (short for Data Latch) has a data input and a clock input. When the clock input is set to true, the output matches the data. When the clock input is set to false, the output retains its current value regardless of the data input.

How can this behavior be created? The D Latch can be constructed using an SR Latch. First, assign the data line to the set input, and the inverse of the data line (using a NOT) to the reset input. In this way, the data input will set or reset the latch as appropriate. Next, apply an AND gate to each of the SR Latch's inputs, with one input of the ANDs connected to the clock. In



this way, if the clock is false, both inputs to the SR Latch will be false and it will hold its existing value.



A stateful truth table describes this behavior.

D	Clk	Q_{prev}	Q
T	T	-	T
F	T	-	F
-	F	T	T
-	F	F	F

In the example of the D Latch, the memory circuit will latch the input data for the entire duration that the clock input is true. This still leaves open the possibility for the value to fluctuate on a single clock pulse. Therefore, a revised circuit is needed which acts only at a single moment: the edge in which the clock changes from one value to another.

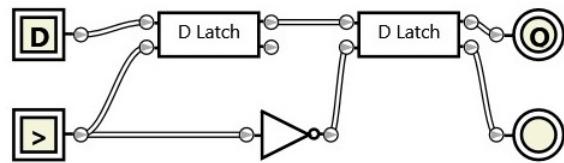
In order to cause a memory circuit to obtain a value only on the clock edge, two latches can be placed in sequence, so that one acquires the value as the clock is true, and the other acquires from the first when the clock switches to false.

■ **Example 17.12** • An edge triggered D Flip-Flop is the most common memory cell design. It has a data input and a clock input. The cell acquires the data input when the clock input drops (goes from high to low). It is also possible to design the flip-flop to acquire the data when the clock input raises.

The technique is to apply the clock input directly to the first D Latch, and then use a NOT gate so the opposite of the clock is applied to the second D Latch. In this

Edge Triggered Flip Flop:
a memory cell which acquires a value only at the moment the clock input changes.

way, then the clock is true, the first D Latch is acquiring the data line. When the clock drops, the second D Latch acquires the value from the first. However, further changes to the data do no effect the circuit because the clock input to the first D Latch is now false, causing it to ignore any data inputs.



A stateful truth table describes this behavior. Note that the phrase “falling” indicates the moment of transition when the clock changes from true to false.

D	Clk	Q_{prev}	Q
T	Falling	-	T
F	Falling	-	F
-	-	T	T
-	-	F	F

■

17.7 Exercises

Solutions to these exercises can be found in Appendix A.16 on page 339.

1. *Problem:* Using the logical expressions for the half adder ($s = a \oplus b$ and $c = a \wedge b$), prove the correctness of the construction of the full adder based on two half adders and an OR gate.
2. *Problem:* Extend the four bit comparison circuit shown in the chapter to have three outputs: one for $a < b$ (existing), one for $a = b$, and one for $a > b$. As much as possible, re-use the existing circuit.
3. *Problem:* A multiplexer is a logic circuit that has n select inputs and 2^n data inputs. The data input corresponding to the given select value is sent to the single output line. Implement a 4 to 1 multiplexer; that is, a multiplexer with 4 data inputs and one output.
4. *Problem:* A clocked T Flip-Flop (T stands for Toggle) has two inputs: T and Clock. It will invert its output if the T is true on the clock edge.
 - (a) Implement a T Flip-Flop.
 - (b) If the T input is held true, the output of the T Flip-Flop should alternate at half the rate of the clock input. Confirm this behavior in your implementation.
5. *Problem:* Implement an addressable memory bank. Provide for four storage locations (addressable by two bits),

with four bits of data held in each storage location. When an address is selected, the current value in that location should be provided. It should also be possible to set a value into the addressed location.

Consider the four bit add/subtract circuit shown earlier in the chapter.

6. *Problem:* Show how the circuit processes the following inputs. Is the result correct or not? If not, why not?
 - (a) $0101_2 + 0011_2$
 - (b) $0101_{2C} + 0100_{2C}$
 - (c) $0101_{2C} - 0001_{2C}$
 - (d) $1110_2 + 1100_2$
 - (e) $1110_{2C} + 1100_{2C}$
 - (f) $1010_{2C} - 1100_{2C}$
7. *Problem:* Devise an improvement to this circuit that will detect if an addition or subtraction result is incorrect due to insufficient bits to represent the result. Assume all values are Two's Complement.
8. *Problem:* Alter the four bit add/subtract circuit. Remove one of the inputs and replace it with four bits of memory. Add an accumulate button which updates the memory to be increased or decreased, as selected, by the amount of the input.

Chapter 18

Finite Automata

Chapter 19

Flowcharts

The techniques seen previously in this part describe specific approaches to solve certain well defined problems. However, a general technique for describing the computational approach to problem solving is needed. Algorithms are used to describe a solution in sufficient detail that it could be implemented using a computer; or analyzed to determine how well a computer would perform while accomplishing it.

Flowcharts are a technique for visualizing the steps a computer would take to solve a problem. In order to compare and contrast different ways of solving a problem, or determine if a proposed solution is correct, or analyze a solution technique for performance, the particular technique must be described in sufficient detail that it could be implemented in a computer. A flowchart helps to ensure that level of detail is met without imposing any undue limitations on the scope of the solution.

19.1 Introduction to Algorithms

Algorithms reflect the fundamental purpose of computing: the intersection between the physical/conceptual machine, and the motivations/goals of the people using them. An algorithm is a recipe, a process, a set of instructions; it describes what steps a



Algorithm: a step-by-step process to solve a certain problem or class of problems.

computer (or human) should take to solve a problem. These steps or instructions can be described in a variety of ways: through diagrams, such as logic diagrams, automata, or flowcharts; through pseudo-code or programming languages, or even in plain language.

Algorithms are commonly encountered in daily life. Cooking recipes are examples of algorithms. A user's manual for some equipment contains algorithms. Most of these algorithms are not specific enough to be processed by a computer, however. Computer algorithms must be completely unambiguous and require no intuition, "common sense" or other interpretation.

Each algorithm solves a particular problem; the correctness of an algorithm cannot be determined without reference to the problem.

■ **Example 19.1** • Given two whole numbers, a and b , find the largest number c for which $a \div c$ and $b \div c$ are both whole numbers.

This problem is known as the greatest common factor problem. Consider the following algorithm:

Algorithm: Return the answer $c = a + b$.

This IS an algorithm, but it does not correctly solve the stated problem. Correctness of an algorithm is almost always the primary determining factor in its consideration: an algorithm which is not correct should not be considered when solving a problem.

Here is another algorithm:

Algorithm:

1. Input a and b as positive whole numbers
2. If b equals 0, return the answer a
3. Save b in a temporary variable t
4. Update b to be the remainder of $a \div b$
5. Update a from t
6. Go to line 2



This algorithm is known as Euclid's Algorithm, and correctly solves the problem stated. However, if the problem were different, this would not be an appropriate algorithm. Algorithm selection and evaluation always depends on the problem at hand.

For any given problem, there could be more than one possible algorithm to solve it. For example, here is a different algorithm to solve the GCF problem:

Algorithm:

1. Input a and b as positive whole numbers
2. Let n be the larger of a and b
3. If $a \div n$ is a whole number, AND
If $b \div n$ is a whole number, return n as the answer.
4. Otherwise, modify $n = n - 1$
5. Go to line 2

This algorithm also solves the GCF problem, but using a different set of steps as compared to Euclid's Algorithm. ■

19.2 Fundamental Control Structures

Looking at the above described algorithms, certain similarities become apparent. Certain words and phrases, like "if" or "go to" appear repeatedly. These concepts are called control structures.

In order to be able to perform general computation (that is, to solve all types of problems that a machine can solve), three basic control structures are needed. The manifestation of these structures can vary dramatically from one programming language or machine to another, but all three will always be present.

1. *Sequence:* Tasks can be performed in an order specified.



In general, any given problem may have many different algorithms that solve it.



Control Structure: a technique which determines how or in what order instructions are processed.

 For a startlingly different computational technique which presents these three structures in a very different way, investigate "Conway's Game of Life".

The flowcharts in this text are created with an older program called the Structured Flowchart Editor, found at <http://watts.cs.sonoma.edu/SFC/>.

 Although a variety of programs will create much more visually appealing flowcharts, the SFC program has the advantage that it enforces correct structure.

 Inadvertently writing complex or ambiguous behavior as a task will make analysis and implementation more difficult, as the precise sequence of steps to complete the task must, at some point, be determined; the flowchart should have accomplished that step.

2. *Selection:* Decisions can be made that effect which tasks are performed, or what order they are performed in.
3. *Repetition:* Tasks can be performed multiple times.

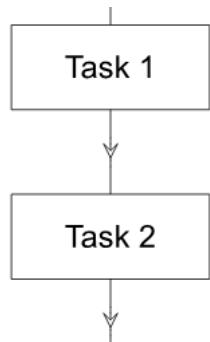
A major innovation in describing algorithms emphasized these three structures and that all components of an algorithm should be explicitly described in terms of these three structures. Thus, instead of using instructions like "go to" to wire instructions together; sequence, selection, and repetition would be connected in predictable ways that made reasoning about the outcome of an algorithm straightforward.

Two famous computer scientists (Abelson and Sussman), once wrote, "Programs must be written for people to read, and only incidentally for machines to execute." This same concern holds for flowcharts: they are primarily communication techniques to describe algorithms, and thus should be as clear and understandable as possible. The use of well-known control structures helps ensure flowcharts are easy to understand.

The most basic unit of a flowchart is a task, which is represented by a rectangle. A task must be something that is unambiguous, with only a single way to do it, and completable in a single step of work. For example, "add a to b" is a good task, but "sort the list" is not (because sorting takes multiple steps, and there might be multiple ways to do it). Likewise, "take the first item from the list" is a good task, but "find the largest number in the list" is not (because finding the largest value takes multiple steps). Simple one-step tasks are known as "atomic", indicating they are conceptually the simplest (or indivisible) kind of thing.

Tasks, or other control structures, can be connected by lines to indicate sequence. A sequence of tasks is normally run from top to bottom, unless arrows or other annotations indicate otherwise. The following diagram shows that task 1 will be followed by task 2. In an actual flowchart, these would be actual atomic tasks.

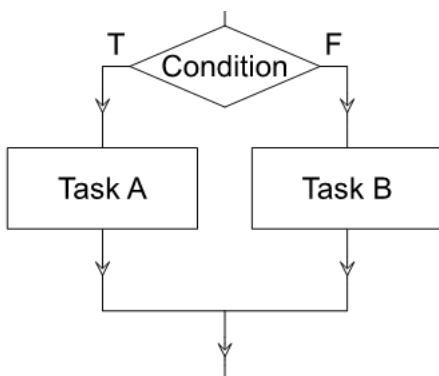




Selection is usually indicated with a diamond shape, or another angled edge. Two lines branch off of a selection, indicating the possible outcomes of a condition. A condition is an expression that evaluates to either true or false; it is not a task to accomplish but a question to answer.

For example, “add 1 to b” is a task, but “is b greater than 1” is a condition. Like a task, a condition should be atomic. A condition such as “is this list in sorted order” is not appropriate, because determining whether or not a list is in sorted order takes multiple steps, and there may be multiple ways to complete the determination.

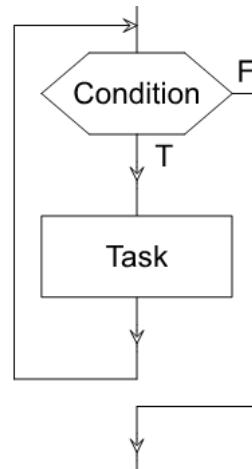
The following diagram shows that task A will be run if the condition turns out to be true, otherwise task B will be run.



Repetition is based on selection, but involves a line that goes back up to instructions previously run. Some programs use a different symbol to differentiate selection from repetition, but in many cases, the same diamond

symbol will be used for both. Like selection, repetition also has a condition. This condition is used to determine if the repetition continues or if the sequence is complete.

The following diagram shows that if the condition is found to be true, task A will be run. The condition will then be checked again, and as long as it turns out to be true, the task will be run. When the condition is found to be false (presumably because of some influence from the task), then the flowchart moves on to whatever follows.

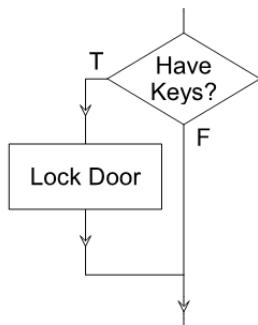


19.3 Selection Control Structures

There are several standard ways to handle selection in a flowchart. The most basic of these is the single decision. The single decision performs some task if a condition is true, but otherwise it skips that task.

■ **Example 19.2** • For example, imagine we are writing a flowchart to describe preparing to leave the house. We might decide to lock the door if we have our keys with us, but not lock it otherwise.

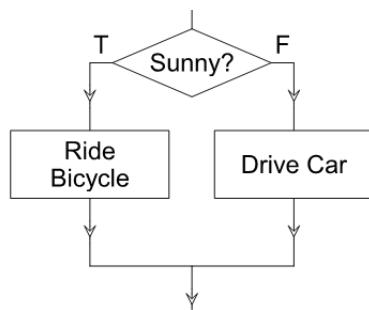




■

A slightly more complicated selection, the double decision, performs one task if the condition is true, and a different task if the condition is false.

■ **Example 19.3** • For example, imagine we are writing a flowchart about going to work. If the sun is out, we will ride our bicycle, otherwise we will take the car.



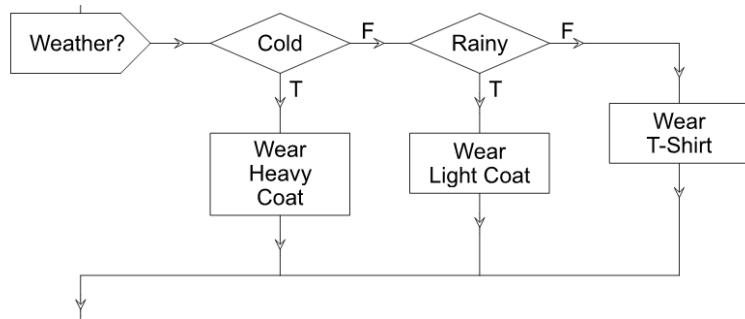
■

The most complex selection structure is the switch, or select-case. This structure allows a single expression to be tested for multiple possible values. Each value has a corresponding task (together, the value and task are known as a case). Finally, there may be a “default” or “otherwise” case whose task is run if no other case matches. Switch structures are difficult to use correctly; it is important that only ONE expression be tested, and each case corresponds to one possible value of that expression.

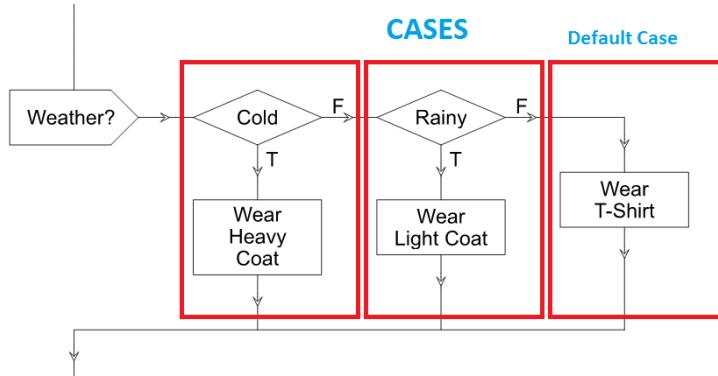
■ **Example 19.4** • For example, imagine we are writing a flowchart about dressing to go outside. If the



weather is cold, we will wear a heavy coat (regardless of whether or not it is raining or snowing). If the weather is not cold but is raining, we will wear a light coat. Finally, if the weather is neither cold nor rainy, we will wear a t-shirt.



In a switch, the first case that matches is the only case that will be run. In this switch example, we first consider if the weather is cold. If it is, we don't also consider if the weather is rainy; we immediately proceed to wearing the heavy coat and then on with the program. Only if the weather is not cold do we then ask if the weather is rainy. All the values being checked relate back to the original expression, "weather". We can divide this structure into pieces to make it easier to follow.



 The default case is not required to have a task; in that situation, the line would still exist but there would be no box, indicated that control simply proceeded to the next structure.

The condition, seen first, indicates what each value will be checked against. We then check each case, from left to right, taking the first case that matches only and running its task. If no case matches (in the above example,



if the weather is warm and sunny), then the default case task is run. Each case can be thought of as a vertical column, and the selection process is deciding which column to run. Since the first column which matches will be run, the default case will always appear last.

19.4 Repetition Control Structures

Selection control structures used conditions to decide which task to run. Repetition extends this idea to repeat a task (known as the loop body) while or until a certain condition is met. It is essential that the task being performed can have some effect on the condition so that the loop can reach termination, allowing the program to continue.

The various types of repetition structures seen in flowcharts are based on where the condition is evaluated: it may be evaluated before the loop body (pretest), after the loop body (post-test), or in the middle.

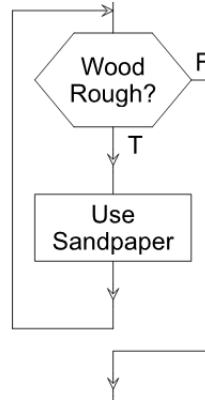
A pretest, or test at the top, style of loop is most commonly seen. This style checks the condition first. Only if the condition is met does the loop body task get run; otherwise, it is skipped. Thus, it is possible for the loop body to never be run at all.

■ **Example 19.5** • In this example, we check if the wood is rough. If so, we sand it, then check again. If the wood was not rough (smooth) to start with, there is no need to sand it.

 *Loop Body:* the task(s) or structures run repeatedly while or until a condition is met.

 *Infinite Loop:* a loop whose body does not change the loop condition, causing the loop to repeat continuously, until the program is manually terminated.

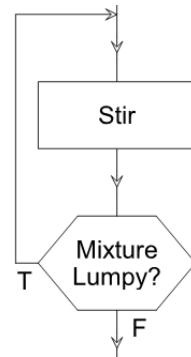
 *Iteration:* a single run through the loop body.



■

A post-test loop is used when the loop condition cannot be checked until the loop body has run at least once. The post-test (test at the bottom) style loop applies when the loop body must always be run at least once.

■ **Example 19.6** • In this example, we need to mix some batter until it is smooth. The check for lumpiness doesn't make sense until the first round of stirring is complete; when the mixture is just poured, no lumps exist. The first lumps don't appear until the batter is first stirred. Thus, we use a test at the bottom loop to ensure the mixture is stirred at least once before checked for lumps.



■

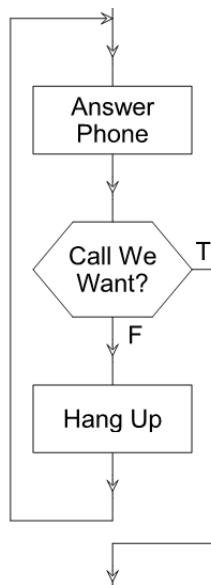
These two styles of loops can be combined with a test in the middle loop style. Test in the middle allows the loop body to be divided up into two portions, with the



condition check occurring in the middle. This form is a generalization of both pretest and post-test loop forms.

■ **Example 19.7** • For example, imagine we are waiting for a very important call (for this example, assume caller ID is not working). In that case, we would need to answer every call. However, if the call received is not the call we are waiting for, we would hang up to continue waiting.

Always pay attention to which branch is marked “T” and which branch is marked “F” when considering any selection or repetition condition.



You may notice in the previous examples the loop either continued when the condition is true, or continued when the condition is false. All three of these loop forms may work either way. A loop that continues when a condition is true is a “while” loop; a loop that continues when a condition is false is an “until” loop. Thus, we could have an “until pretest” style loop, or a “while test-in-the-middle” style loop, or any other combination.

A final repetition style sometimes seen in flowcharts is the counting loop, also called a for loop. The counting loop is used whenever a task needs to run a certain number of times (a number either known in advance, or determined before the loop through other tasks). To perform a loop of this type, a counter is used to control how many times the task is repeated.

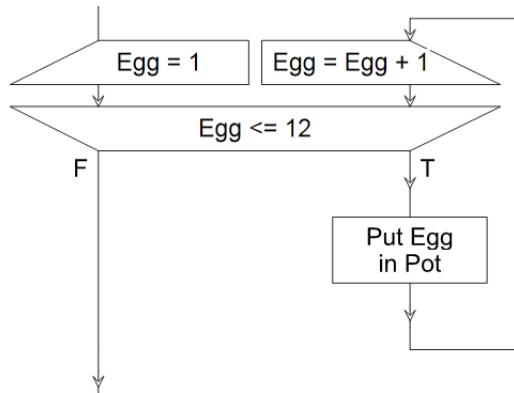


Counter: a variable changed by a fixed amount at each iteration of a loop.



A counting loop is a special case of pretest while loop.

■ Example 19.8 • For example, we want to boil a dozen eggs. The water pot has been prepared, now we need to move each egg individually into the pot. There are known to be a dozen eggs, so the “move egg” task will repeat 12 times.



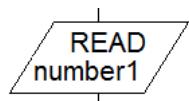
In this diagram, the top left box ($\text{egg} = 1$) is an initialization task that occurs before the loop begins. The larger box ($\text{egg} \leq 12$) is the condition, with the T and F branches coming out of it. After the egg is placed in the pot, the upper right box ($\text{egg} = \text{egg} + 1$) increments the counter before the condition is evaluated again. This increment is technically part of the loop body and is run at every iteration. ■

19.5 Other Symbols

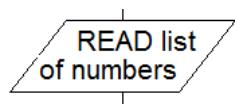
A flowchart must also clearly define what its source data (input) is, and then what results (output) it produces. One way of doing this is with the Input/Output symbol, represented as a parallelogram. This symbol is used whenever values need to be transferred in or out of the flowchart. Input values are specified with “READ” or “INPUT” phrases, along with a variable name or description of where the result will be stored. Output values are specified with “WRITE” or “OUTPUT” phrases, along with the variable name that has been prepared earlier in the flowchart and contains the desired result.



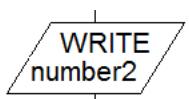
Here are several examples of Input/Output symbols. The first one shows a single number being stored into the variable number1.



It is also possible to handle lists or values, which can be stored into a list variable. Operations on list variables will be discussed in the next section.



Finally, results produced during the operation of the flowchart are returned as result values.



■ **Example 19.9** • For example, consider the case of finding the maximum of two numbers. We will need to input the two numbers and then compare them to find out which number is larger. There are three possible cases:

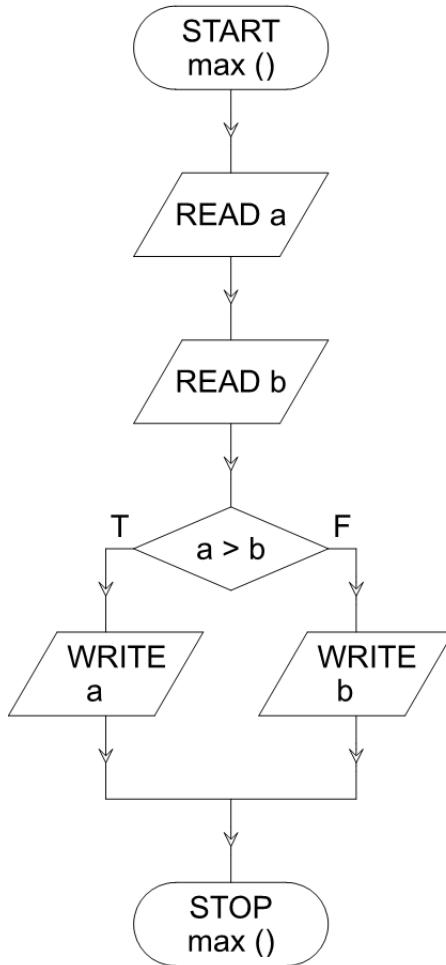
1. The first number is larger.
2. The second number is larger.
3. The two numbers are equal.

The third case is sometimes called a “boundary case”, “edge case” or “corner case”, indicating that it is a situation on the edge of the usual defined behavior. When finding the larger of two numbers, what do we do if the two numbers are equal?

It is important when developing an algorithm to consider all corner cases. The act of breaking the algorithm

 **Corner Case:** occurs when system input is technically legal but unexpected, at the edge of an allowable range, or otherwise unusual.

down into flowchart level (or psuedocode) can help discover such cases. In this example, when the numbers are equal, we will return either value (since they are the same). This allows us to roll the equals case into either of the larger cases, and the algorithm need only handle two cases (when the first number is larger than the second number, and “otherwise”).



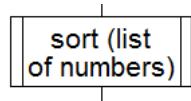
Dozens of other symbols exist for use in specialized flowcharts.

The flowchart combines input, decision, and output to complete a specific task. ■

As mentioned earlier, each task block should be atomic. However, once we have defined some process (such as how to sort a list), we may want to be able to reference that operation multiple times. The rectangle with extra bars on the sides is used to indicated a composite task; a task that is not atomic but has been previously



defined (either in flowchart or discussion) so there is no ambiguity as to what and how the procedure is performed.



At this point, it is becoming clear that flowcharts can be confusing and difficult to understand. Fortunately, explanations can be embedded into flowcharts. These explanations, called comments, do not indicate any behavior but merely describe the thought process behind some set of tasks, or indicate why a particular technique was chosen. To indicate that they do not effect the outcome in any way, comments are indicated off to the side of the main task flow line.

Explanation
of goal or
process

19.6 Managing Lists

A sequence of values in the form of a list is a common basis for algorithmic problems. Many algorithms operate on arbitrarily sized lists. There are two main approaches to handling list input: the entire list can be input up front and then handled piecemeal, or each value in the list can be input one at a time. Some algorithms work better with one technique rather than the other.

When a list is read in up front, the list is managed by manipulating selected elements in the list. The advantage of this technique is that elements in the list need not be read or manipulated in any particular order. The disadvantage is usually increased complexity.

The list will be described with some variable (often just `list`) and specific elements will be accessed by index (position) in the list. The first position is usually 0. So

$\sum_{n=1}^{\infty} \frac{1}{n^2}$

A list is different from a set in two ways: a list has ordering (first value, second value, etc.) and a list may have duplicates.

Although this text will focus only on lists for input, the same two techniques can be used to construct lists for output.

the first element of the list is referred to as `list[0]`. A variable such as `count` or `size` may be used to indicate how many items are in the list. Given that the first item has index 0, the last item will have index of one less than `count`. So a list with five elements (`count = 5`) will have indices 0, 1, 2, 3 and 4. We could reference the last element in a list as `list[count - 1]`.



End of File: abbreviated EOF or EOS (end of stream), a Boolean condition that indicates when no more values remain to be read from a list.

On the other hand, the list can be handled by performing input within a loop. In this case, no overall “list” variable is ever constructed. Instead, a loop continues as long as more items remain in the list, inputting one item at a time. A special variable called EOF (end of file) or EOS (end of stream) is used to indicate when there are no more items in the list. The loop continues until EOF is true (or while EOF is false).



Nested Loop: a loop within the body of another loop.



Accumulator: a variable which stores intermediate results of an aggregation, usually a sum.

In some cases, items in a list may need to be compared to each other. Such a circumstance calls for a nested loop. It is also common for a loop to add up or otherwise aggregate the values in a list. A variable which builds up a sum or other aggregation is called an accumulator.

■ **Example 19.10** • For example, consider extending the earlier example of finding the maximum of two values to finding the maximum value in a list. If the entire list is input up front, then a counting loop over the indices of the list can be performed (from 0 to `count - 1`, as described). In each iteration of the loop, the current value in the list will be compared to the maximum found so far, and if the current value is larger, it will replace the maximum so far.

For this to work, some initial value of maximum so far is needed. An initial guess might be to propose 0 as an initial maximum, since it will be replaced by the larger numbers as the algorithm progresses. However, what if the list consists entirely of negative numbers? In that case, an initial value of 0 for maximum will cause 0 to be incorrectly returned as the actual maximum (since 0 is larger than all values in the list, it will never be replaced). Thus, the only correct action we can take is to select an actual value from the list to be the initial value for maximum.



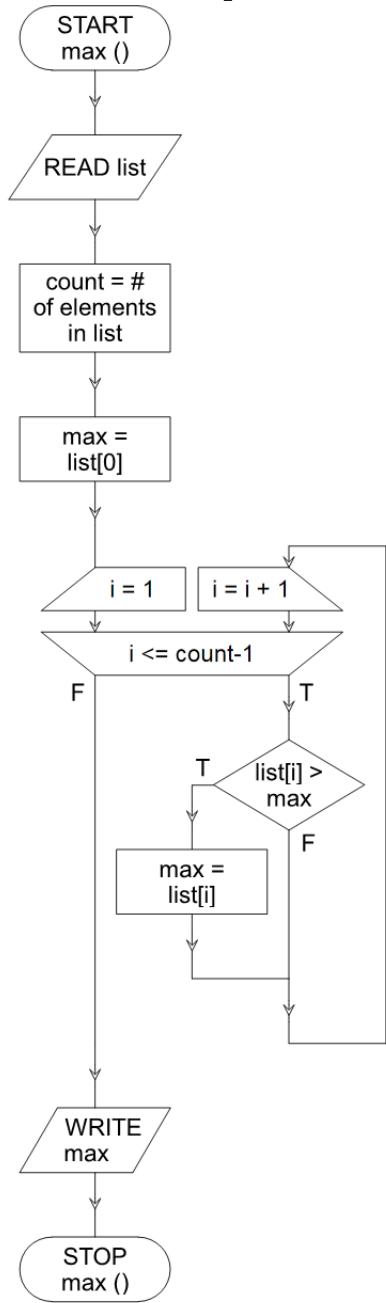
Recall that each item in the list can be accessed by index (position), with the first item having index 0. In this flowchart, we set the initial value of max to the first item in the list `list[0]` and then loop through all remaining items with a counting loop. We inspect each item of the list one at a time using the counter variable `i`, so that `list[i]` refers to the particular item being inspected (depending on the value of `i`, it could be the second, third, fourth, fifth, ... item).

Another way to approach this problem is to read each value one at a time. In this case, a loop will check if more values remain to be read. If so, the next value will be read and compared to the current max, with the max being updated if the read value is larger. As in the previous version, the first value will be read before the loop begins to establish a baseline max.

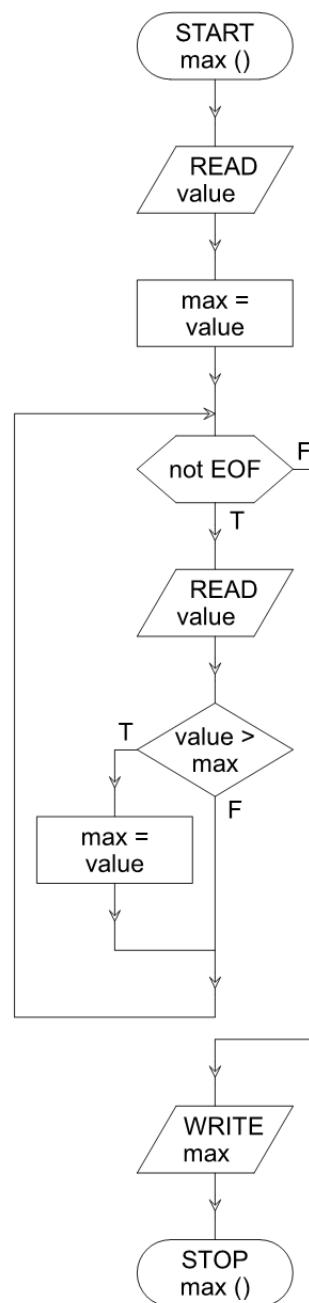
This version does not require any loop indices, so it could be considered simpler. However, some operations (particularly those that aren't a linear single pass through a list) require that the entire list to be read in advance, so being familiar with both techniques is appropriate.

Here both techniques are implemented as flowcharts and compared side-by-side.

Read whole list up front



Read one element at a time



19.7 Functions and Reuse

In larger flowcharts, it is common to find sub-tasks which may be needed in several places, or could be conceptually extracted and used in other solutions. In order to avoid overly complex flowcharts, it is best to first define these subtasks, and then refer to them in the main solution.

■ **Example 19.11** • For example, imagine we want to take a list which has duplicates and produce a new list of the same values but without duplicates. There are several ways to accomplish this task. One straightforward way is to start with an empty new list, and take the input one value at a time. For each value in the input, check to see if it exists in the new list. If not, add it to the new list.

In order to fully define the solution, we must understand how to check to see if a value exists in a list. This is a smaller problem which could be solved by itself in preparation for solving the larger problem.

To determine if a value exists in a given list (a problem we'll call "contains"), we can loop through each entry in the list and check if it matches the desired value. If so, a Boolean variable (acting as a flag) will be set to true indicating the value has been found.

With the definition of "contains" so determined, we can construct a solution to the original problem (removing duplicates) by reference to the procedure contains where needed. Dividing the original problem into smaller parts makes it easier to solve and makes each solution part easier to understand.

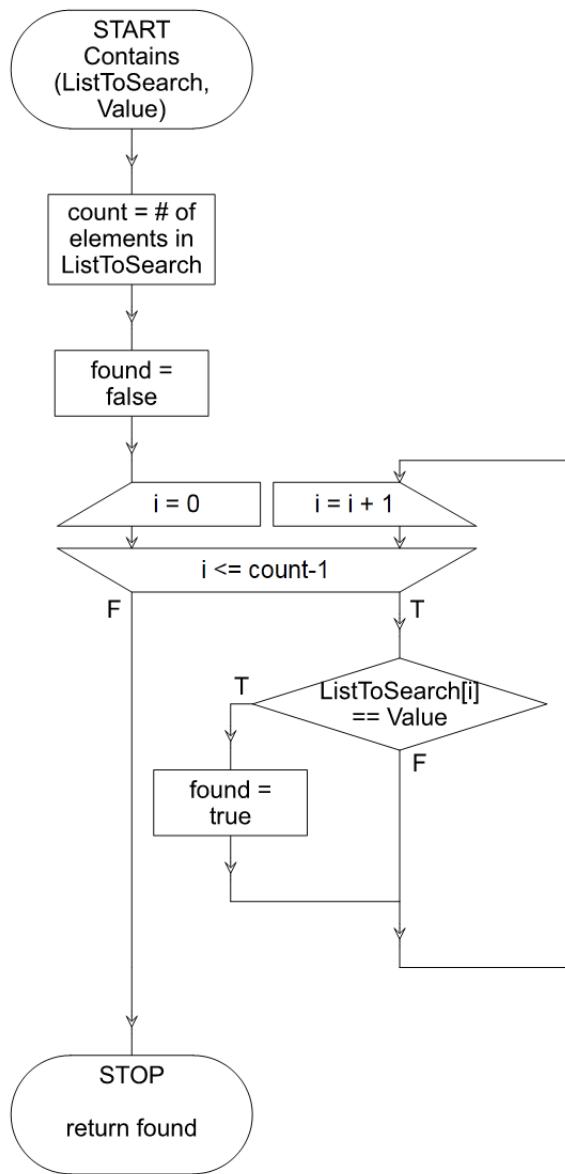


Breaking down a large problem into smaller discrete pieces is almost always a good technique.

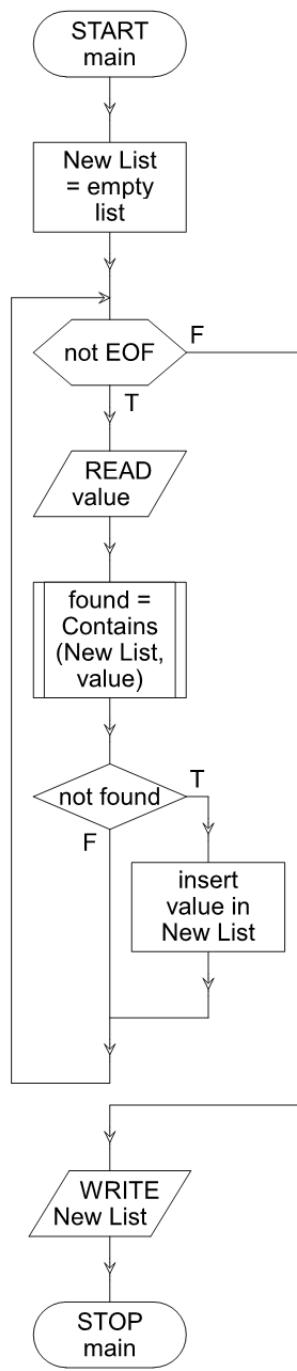


Flag: a Boolean variable which indicates if a certain condition has occurred.

Flowchart for "contains"



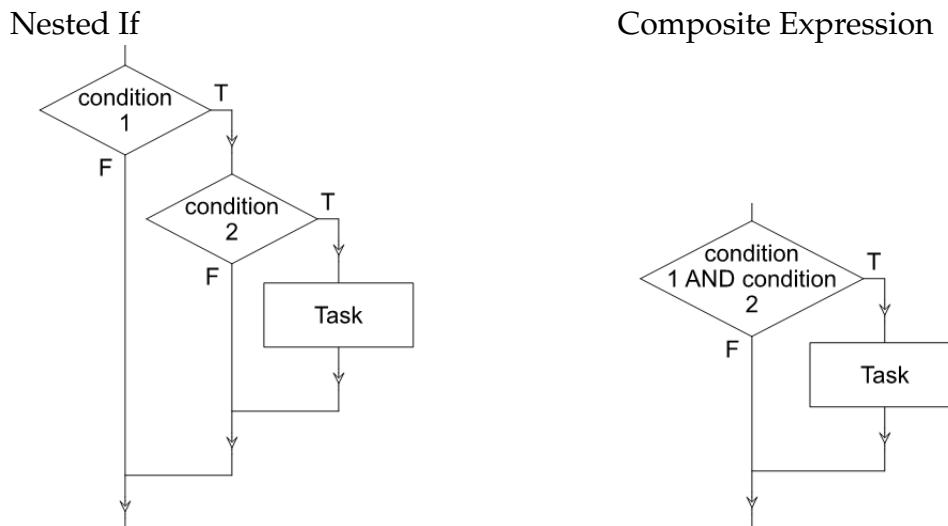
Removing duplicates using "contains"



19.8 Logical Expressions and Set Extraction

Nested decisions may sometimes be transformed into a single logical expression, or vice versa. The most reliable way to analyze these decisions is to create a truth table indicating when a task is executed, and then derived a simplified logical expression from that truth table.

■ **Example 19.12** • The two decisions below are equivalent.



We can confirm this equivalence by creating a truth table which shows under what conditions the task is executed.

Condition 1	Condition 2	Task
T	T	T
T	F	F
F	T	F
F	F	F

This truth table shows that the AND of both conditions is sufficient to decide when to execute the task. ■

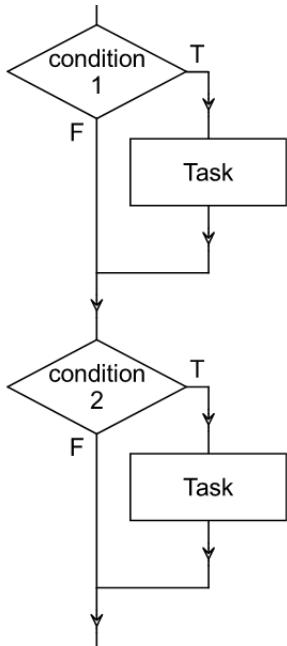
However, we must be careful to ensure that the task, when executed, is executed the same number of times



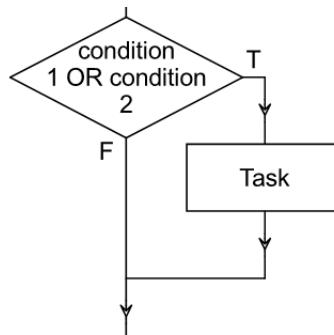
in both cases. Or, if we can conclude that multiple executions of the task are irrelevant (for example, if it is just setting a flag), only then can we ignore multiple executions.

■ **Example 19.13** • For example, the following two flowcharts are NOT guaranteed to be equivalent. The reason is that, if condition 1 and condition 2 are both true, one of the flowcharts executes the task once, and one of the flowcharts executes the task twice.

Both true: executes Task twice



Both true: executes Task once

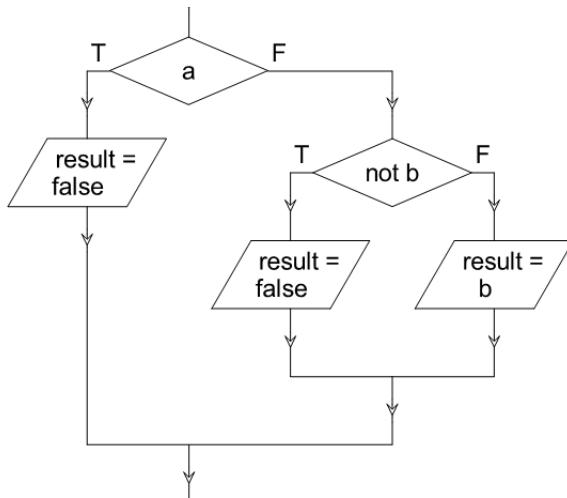


■ However, if the task is simply setting a flag or something else that can be repeated without concern, then the left flowchart could be replaced with the right flowchart, and vice versa.

The easiest flowcharts to simplify are those that depend entirely on Boolean inputs and set Boolean flags as their only tasks. In this case, the flowchart can always be reduced to a single logical expression for each flag set.

■ **Example 19.14** • For example, consider the following flowchart.





There are two variables (a and b) used in this flowchart. The task sets the Boolean variable `result`. Since all variables are Boolean, we can directly create a truth table from this flowchart. In the table below, `result` is shortened to r .

We start by setting up the truth table based on the variables in use.

a	b	r
T	T	
T	F	
F	T	
F	F	

Next, the left branch tells us that when a is true, then r will be false, regardless of b .

a	b	r
T	T	F
T	F	F
F	T	
F	F	

What if a is false? Then a further decision is encountered: $\neg b$. Be careful with this one. The true branch will be followed when $\neg b$ is true, which only occurs when b itself is false. Thus, when b is false, r will be false.



a	b	r
T	T	F
T	F	F
F	T	
F	F	F

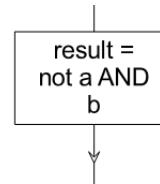
Sometimes it is better to leave more structure in the flowchart to provide space for comments and also to decrease the amount that a reader needs to understand at one time. The final level of simplification is a design decision.



In the remaining case, r will be equal to b , according to the flowchart.

a	b	r
T	T	F
T	F	F
F	T	T
F	F	F

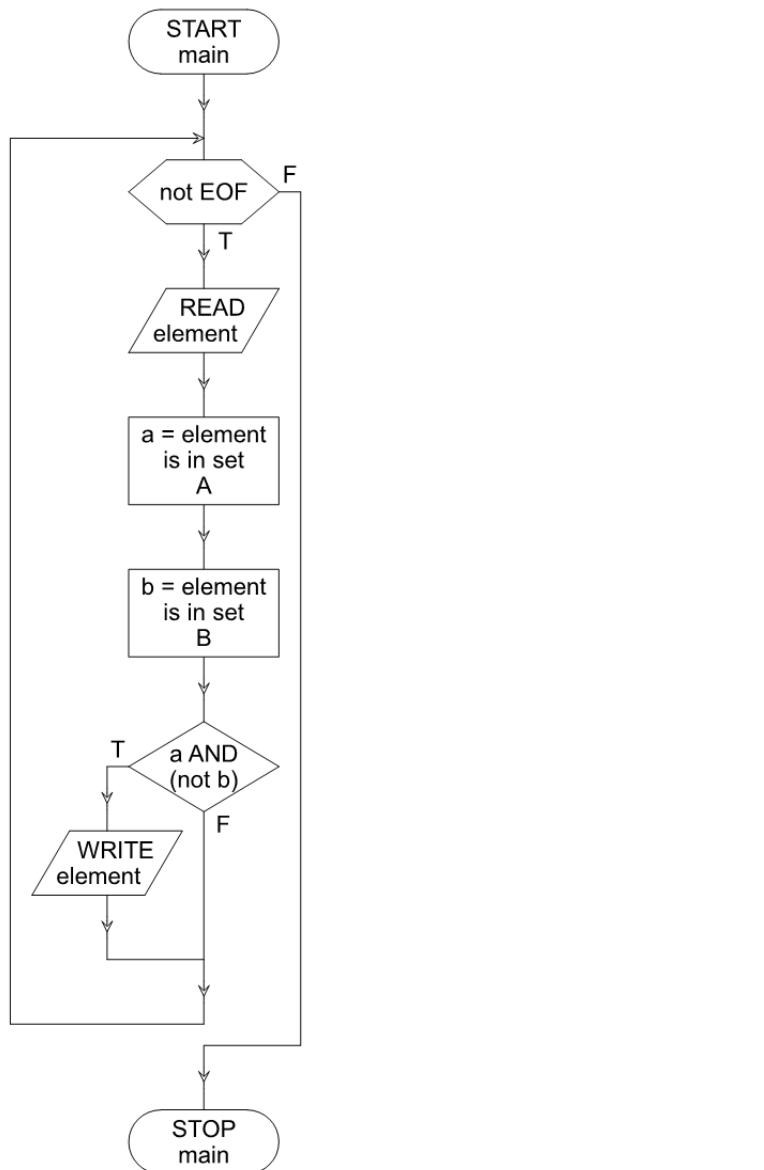
This truth table can be represented as the logical expression $\neg a \wedge b$. Thus, it can be entirely replaced with a single task based on this expression.



A similar technique can be used to determine which elements exist in a given set expression. In order to extract these elements, a loop over the universe of elements is assumed. Next, for each set involved, a Boolean variable is created indicating if the element is a member of that set. Finally, a logical expression corresponding to the set expression is used to determine if the element is a member of the set or not.

■ **Example 19.15** • For example, consider the set expression $A \cap B'$. The equivalent logical expression is $a \wedge \neg b$. Using a loop over the input, which is assumed to be the universe, the following flowchart outputs all elements in the set.





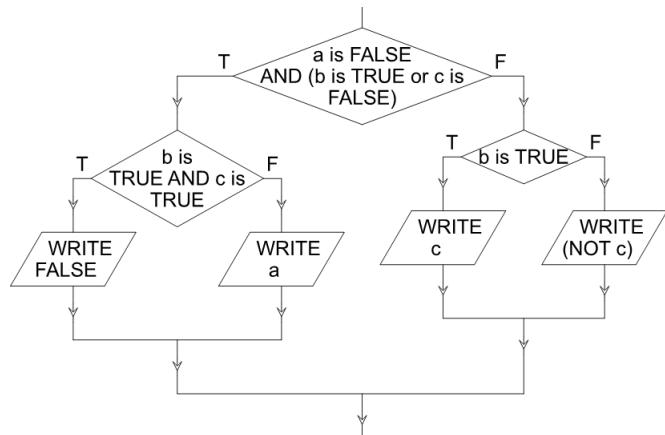
19.9 Exercises

Solutions to these exercises can be found in Appendix A.17 on page 355.

- Problem:* Convert the 12 eggs counting loop example into pretest while loop style.
- Problem:* Write a flowchart that calculates the average (mean) of a list of numbers.
- Problem:* Write a flowchart that determines whether or not a given list of numbers is in sorted order from smallest to largest.
- Problem:* Write a flowchart that, given a list, outputs the list in reverse.
- Problem:* The Fibonacci sequence is a list of numbers, where each number is derived by adding the previous two together. The first two numbers in the sequence are 1. The first eight Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21.
Write a flowchart that given a position number, outputs the corresponding Fibonacci number. For example, if the position is 1 or 2, the Fibonacci number is 1. If the position is 6, the Fibonacci number is 8, and so on.
- Problem:* Consider the Fizz-Buzz problem, a small programming challenge

proposed for use in job interviews. Write a flowchart that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

- Problem:* Assume a , b and c are Boolean variables; determine a logical expression for the following flowchart.



- Problem:* Create a step-by-step version of the flowchart from the previous problem, using only one variable per decision condition (in other words, no AND or OR).

Chapter 20

Analysis of Algorithms

The purpose of using computers is to solve problems, preferably more accurately and faster than a human could solve them. For any given problem, there could be multiple different solution techniques. Given a single problem and many proposed solutions, how can one solution be selected? Several factors may be involved in such an evaluation:

- Cost to implement (time/expertise)
- Robustness (can it handle all the inputs?)
- Generality (can it solve other problems as well?)
- Maintainability (can it adapt to changes in the problem?)
- Performance (how fast is it? how much memory or disk space does it need?)
- And others...

Many of these considerations are part of a design and systems analysis process. These considerations are very important, but are largely people-driven and fall outside the scope of this text. A review of Requirements Analysis and/or Systems Analysis is recommended to the interested reader.

For our purposes we will consider only the measure of performance. Given a computational problem, it is possible to devise multiple algorithms to solve it. Some algorithms may run faster, or take less memory, than others. Algorithms which run faster and take less memory can process larger amounts of data on the same hardware, so it is advantageous to select them when possible.

20.1 Performance Factors

Many factors influence the performance of a particular algorithm, on a particular computer, on a particular set of data. For example, a faster computer might complete the same algorithm on the same data in less time than a slower computer. A computer running many tasks at once might complete the same algorithm on the same data in a longer time compared to a similar computer whose only purpose is to run the algorithm. The identical computer might even complete the identical algorithm with identical data in slightly different running times, thanks to different CPU caching and operating system behavior.

When evaluating algorithms for performance, we want to exclude from consideration physical and environmental factors which would influence any algorithm put in the evaluation spot. It would not be beneficial to compare two algorithms by running one on a slow computer and the other on a fast computer. That evaluation is more likely to show which computer is faster rather than to actually evaluate the algorithm.

In order to avoid biased measurements, algorithm performance is not usually measured in terms of fixed time, such as seconds; in a few years, a faster computer will perform the algorithm in fewer seconds, so such a measurement is not long-standing. In addition, the measurement of seconds (or any other fixed time), is influenced by environmental factors such as other programs running, the programming language used to implement the algorithm, and component speed (memory speed, cache speed, etc.).



20.2 Complexity Classes

Instead of time units, performance is measured using complexity classes. Complexity classes relate the size of the input (how much work the algorithm is assigned) to how many steps the algorithm will take to complete this work. This relationship is true regardless of environmental factors or hardware. This result is categorized into one of several broad complexity classes, which emphasize which algorithms have very different performance while grouping together those whose performance is likely to be quite similar.

The emphasis in complexity classes is: as the input size grows, how much longer does the algorithm take to complete? In the era of massive datasets, millions of transactions per second, and an ever-growing need to manage data in real time, the performance of algorithms on very large input sizes is important. If we know an algorithm's complexity class, we can predict how it will respond on a very large dataset, even if we have never run the algorithm.

One difficulty with complexity classes is defining input size. For an algorithm which sorts data, the input size is straightforward: how many items are there to sort? For an algorithm which searches documents, the input size could be trickier. It could be how many documents exist to be searched, or the total length of all documents to be searched, or the number of search terms in use, or some combination thereof.

Complexity classes are also represented in terms of what scenario they represent (best case, average case, or worst case being the most common). We will generally focus on worst case scenarios, and represent complexity using Big-Oh notation, which shows only the most significant term.

Imagine a stack of unsorted papers. We are looking for a particular one. An algorithm to find this paper, if it exists, is to start at the beginning, and look at each paper in order, until either the end is reached or the paper in question is found. This algorithm is known as linear search. In the best case, the paper might be right on

 *Complexity Class:* formula describing how the running time (or memory usage) of an algorithm changes relative to changes in the size (or quantity) of input.

 Defining input size is important. The complexity class of an algorithm depends on the definition of the input size, and may change if the definition of input size is changed.

 *Big-Oh:* short for “biggest order”, indicates the most significant complexity term, with coefficients dropped.

 *Linear Search:* checking each element in a sequence one at a time, from beginning to end, until the desired element is found or the end is reached.



An algorithm whose worst case running time is representable by $O(n^a)$ for any constant a , is a polynomial time algorithm.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

top: in which case only 1 step was needed. In the average case, the paper might be somewhere in the middle, in which case we would have to look through about half of the papers to find it. This takes $\frac{n}{2}$ steps if there are n papers (notice here we have defined number of papers as the input size). In the worst case, the desired element will be the last one we check, or it might not be present at all. In this case, we have to check all the papers. Focusing on worst case, we would say that the algorithm linear search is $O(n)$, because up to n steps may be needed to find the result.

Complexity classes are often divided into two categories: polynomial, and worse than polynomial. Algorithms which exhibit worse than polynomial running times are usually considered intractable, that is, unusable on datasets of any practical size.

Here are some common complexity classes with names. The top part are polynomial classes; the bottom part shows worse than polynomial classes.

Name	Big-Oh Notation	When the Input Size Doubles, the Time Taken...
Constant	$O(1)$	stays the same
Logarithmic	$O(\log n)$	increases slightly
Linear	$O(n)$	doubles
Quadratic	$O(n^2)$	quadruples
Cubic	$O(n^3)$	is eight times longer

Name	Big-Oh Notation	When the Input Size Increases by 1, the Time Taken...
Exponential	$O(2^n)$	doubles
Factorial	$O(n!)$	is n times longer

20.3 Binary Search: A Practical Example

Imagine we have an application that, at some point, receives a list of sorted values. The application wants to determine if a certain query value is or is not present in the list. How can this be accomplished? Two algorithms come to mind: we could use the linear search algorithm, discussed previously. Alternatively, because



the list is already sorted, a binary search could be employed.

A binary search works on a sorted list by first examining the middle element in the list. If the middle element is the desired result, the search stops. If the middle element is less than (appears before in sorted order) the desired result, the bottom half of the list is discarded and the procedure repeats. If the middle element is greater than (appears after in sorted order) the desired result, the top half of the list is discarded and the procedure repeats. At each step, half of the remaining list is thrown away, until the result is found or no more list remains.

Consider the following example of binary search in action. Assume we are looking for the number 3. At each step, the middle element will be selected, and then half of the list will be discarded. When only two items remain, they will be compared individually in sequence.

1	3	4	4	5	6	6	7	8
1	3	4	4	5	6	6	7	8
1	3	4	4					
1	3	4						
1								
1	3							
1	3							

The result was found in six steps (if comparing and discarding count as two steps). Performing this same search using the linear search algorithm yields:

1	3	4	4	5	6	6	7	8
1	3	4	4	5	6	6	7	8

The result was found in only two steps! Does this mean the linear search algorithm is our algorithm of choice for the application?

Not necessarily. We have shown that the linear search algorithm performed better in one specific case. However, we need to know what the worst case scenario

 *Binary Search:* a search technique for ordered lists which cuts the search space in half at each step.

 Algorithm performance at fixed input sizes is irrelevant to complexity analysis. Only the change in time or space needed as the input size grows is considered.





The times shown here are actual measured times. These times do not exactly follow the complexity classes of the algorithms due to the aforementioned environmental factors.

is as the input size grows. It is unlikely that our application, if it enters heavy use, will be content with searching lists of less than ten elements. How will these searches perform on lists of a million elements, for example?

Running an actual test case on a computer is instructive, although we must be careful of environmental impacts, and the limited ability of computers to measure very quick events. Running these two techniques on a million elements generates a time of 0.019 seconds for the linear search, and 0 seconds for the binary search. Of course, the binary search didn't really complete in zero seconds, it was simply faster than the computer was able to measure.

Both of these times seem very, very fast. So is there any reason to choose one search over another, from a performance point of view?

Increasing the input size to 10 million yields an interesting result. The linear search now completes in 0.352 seconds, whereas the binary search still shows 0 seconds. If 100 million elements need to be searched, the linear search now takes 3.555 seconds, whereas the binary search still shows zero.

These examples were performed using integer numbers, which are very easy for a computer to compare. If the lists consisted of some more complex component, which took much longer to compare, the results would be even more dramatic: as the input size increases, binary search remains very fast, whereas linear search time increases with the input. A complete binary search of one million elements takes only 27 divisions, whereas a linear search of that same space could take up to one million steps.

Linear search was already concluded to be $O(n)$; binary search is $O(\log n)$, a better complexity class, which shows faster times as the input size increases.



20.4 Step Counting

Given an algorithm, how can the time complexity be found? It is possible to measure actual times taken, and fit them to one of the curves. However, this technique is quite error prone. A cleaner technique, although one with its own limitations, is step counting. In this technique, an algorithm is run by hand with several small inputs, and a relationship between input size and number of steps is established.

In order for step counting to work properly, each “step” (or action) must itself be constant time, or atomic. If we perform a large and input-dependent operation, like “sort the list” or “find the smallest number” as one step, the count will not accurately reflect the algorithm’s time complexity.

■ **Example 20.1** • Consider the example of binary search. A binary search of two elements:

a	b
a	b

Takes at most two steps. A binary search of three elements:

a	b	c
a	b	c
a		

Takes at most three steps. A binary search of four elements:

a	b	c	d
a	b	c	d
a	b		
a	b		

Takes at most four steps. A binary search of five elements:



a	b	c	d	e
a	b	c	d	e
a	b			
a	b			

Takes at most four steps. This is the first indicator we have that binary search may be better than linear time!
A binary search of six elements:

a	b	c	d	e	f
a	b	c	d	e	f
a	b	c			
a	b	c			
a					

Takes at most five steps. Jumping forward, we also found that a binary search of eight elements was completed in six steps.

Thus, the number of steps required for a binary search can be closely approximated by $2 \log_2 n + 2$ for n elements. When translating this value into Big-Oh notation, only the largest term is kept, and any coefficients are dropped. Therefore, binary search has $O(\log n)$ time complexity. ■

20.5 Loop Analysis

 Be sure to inspect the algorithm for hidden complexity, like sorting, that may incur a substantial time complexity.

 Sorting incurs $O(n \log n)$ time complexity.

An alternative to time consuming (and sometimes ambiguous) step counting is loop analysis. In order for the time an algorithm takes to increase with increased input size, the increased input size must result in additional work. This work often appears in the form of repetition, such as loops. By inspecting an algorithm for repetition, the time complexity can often be found.

Care must be taken when analyzing an algorithm in this manner to ensure that all operations are accounted for. As in step counting, each action should be constant time and atomic.



To perform a loop analysis, look for repetition in an algorithm. Any part of an algorithm which is repeated forms a loop. This can be clearly seen on flowcharts with “circular” regions. An algorithm which consists of no repetition is constant time, $O(1)$. Repetition which is not based on the size of the input (for example, “repeat this operation five times”) is also constant time: as the input size grows, the number of steps will not change.

Most repetition, however, is based on input size. A loop which reduces remaining input by a constant (usually 1) at each step is a linear time, $O(n)$ loop: it will perform the inner body once for each block of input. In order for the algorithm to be entirely $O(n)$ the contents of the loop body must run in constant time. If there is a loop (based on input) within a loop, then the algorithm is likely quadratic $O(n^2)$.

If the loop cuts the input in half at each step, then doubling the input size is required to add an extra iteration to the loop. This results in $O(\log n)$ time complexity. Binary search divides the input in half at each step, and has $O(\log n)$ time complexity.

Loops which make less progress, or generate more work as they go along, may be in the higher time complexities.

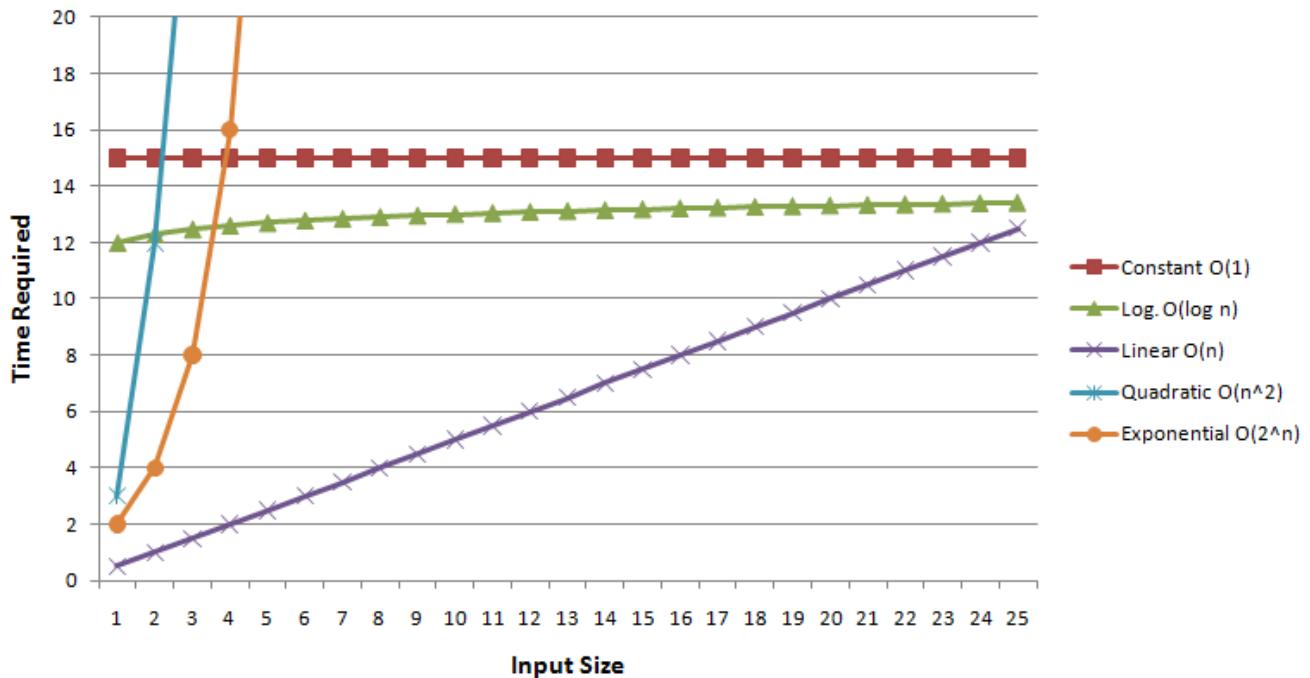
Loops performed in sequence (one after the other) are addition, and so only the worst loop will decide the algorithm’s time complexity. For example, imagine an algorithm which first had a loop that stepped through each item in the list one at a time, next it performed a nested loop, then another nested loop, and finally a binary search. The time complexity of that algorithm is $O(n + 2n^2 + \log n) = O(n^2)$. The smaller terms fall away, and only the most significant term (without coefficients) remains.

If all loops are linear on the input, then nested loops k levels deep have $O(n^k)$ time complexity.

20.6 Comparison of Complexity Classes

How much worse is a quadratic algorithm compared to a linear algorithm? The answer is “it depends”. Time complexity tells us only how the time taken changes as the input size increases: it does not guarantee which algorithm will be faster at a certain input size. With small inputs, the quadratic algorithm could even be faster. As the input size grows, however, the quadratic algorithm will soon take longer than the linear algorithm.

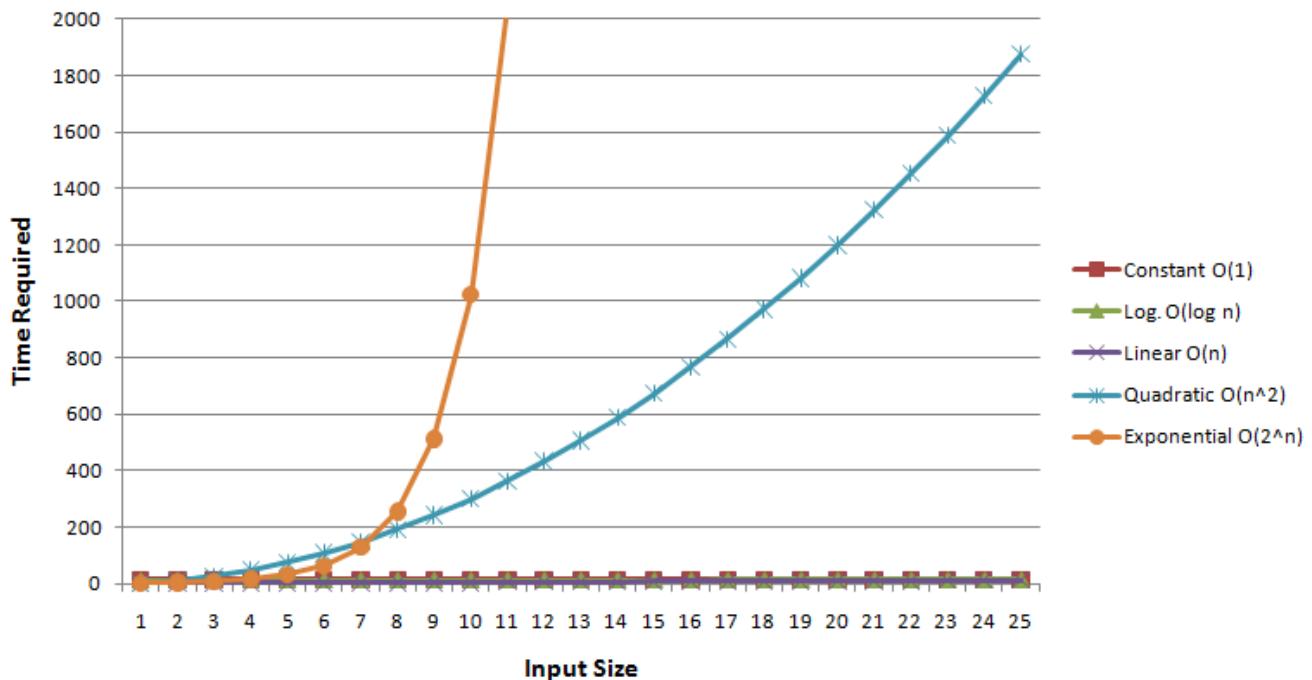
The following graph shows examples of time taken for an algorithm of each class. Notice that, for small inputs, there is no clear relationship between time complexity and time taken. However, as the input size increases, the worse time complexities take longer.



In the above graph, it is clear that quadratic and exponential algorithms become very expensive (take a long time) even with small increases in input size. To appreciate this cost, and to compare these two classes, the same graph is presented again. In this case, we



have adjusted the “time required” axis only. Notice all the sub-quadratic time complexities blur together, and quadratic and exponential are both dominating, with exponential quickly exceeding even the increased chart bounds. By the right-hand edge of the chart, the exponential algorithm incurs over three *million* steps.



It is important to distinguish an algorithm from a problem. Algorithms have time complexities; problems do not. However, for any given problem, there is usually a “best” known algorithm. The time complexity of a problem is often given as the time complexity of the best known algorithm; however, it is possible in the future that better algorithms could be devised and reduce the time complexity for the problem.

Programmers run afoul of time complexity in two common cases: sorting and optimization. A variety of sorting algorithms exist, and many have a worst case time complexity of $O(n \log n)$. Rather than use well designed sorting algorithms built-in to a language, programmers sometimes re-implement their own approach to sorting, often poorly.

■ Example 20.2 • Consider the following algorithm:

A full discussion of the time complexities and techniques of various sorting algorithms, provided with animated visualizations, can be found at <http://www.sorting-algorithms.com/>



The complexity of $\sum_{n=1}^{\infty} \frac{1}{n^2}$ $O(n \log n)$ is slightly worse than $O(n)$ but much better than $O(n^2)$.

1. Start at the first item in the sequence.
2. If this item belongs after the next item, swap them.
3. If there are more items, go the next item and step 2.
4. Otherwise, if any swaps have been made since step 1, go back to step 1.

This algorithm performs a loop over all inputs, but repeats that loop potentially many times (until everything is sorted). An item, at most, would have to move from the start to the end to be fully sorted, so each loop could occur up to n times (where n is the number of elements in the input). Nested loops, each dependent on the input, give $O(n^2)$ time complexity. ■



The moral of the story is: unless you really know what you're doing, stick with the built-in language sorting features.

The time complexities of $O(n \log n)$ and $O(n^2)$ may sound similar, but $O(n \log n)$ is much closer to linear than to quadratic; compare the linear and quadratic time lines on the previous graph.

Sorting algorithms with poor time complexities may still be chosen in certain cases, depending on the application. For example, the algorithm shown above performs reasonably well (near $O(n)$) if the input is nearly sorted and needs only a few minor corrections. Quicksort, which has a worst case time complexity of $O(n^2)$ is very popular because its actual performance time on real world data tends to beat sort algorithms which have better theoretical limits.

Optimization is another area where programmers run into trouble. Most algorithms that solve optimization problems are both fairly simple to write and very bad in time complexity.



Traveling Salesman Problem: an optimization problem which requests the ideal route between a series of locations.

■ **Example 20.3** • Consider the case of a delivery company which wants to minimize the amount of driving necessary to deliver packages. Each day, a list of delivery addresses is available; as are the distances between addresses. An algorithm to find the shortest sequence of addresses would be nice.



A straightforward approach is to consider all possibilities: For the first stop, there are n possibilities. For the second, $n - 1$, and so on. This gives a total of $O(n!)$ possible routes; to consider each route and find the shortest then takes $O(n!)$ time; worse than exponential! Applying more insight to the problem produces an improved algorithm, but even the best known algorithm for this problem is $O(2^n)$. This particular optimization problem, known as the Traveling Salesman Problem, is frequently encountered in a variety of optimization situations. The best-known time complexity for a solution to this problem means, in practice, that finding an optimal solution is almost always impractical. ■

More complicated yet computationally practical approaches involve heuristics. Heuristic algorithms do not attempt to find the optimal solution; they find a good solution (for various definitions of “good”) in much faster times. Considering the delivery driver example, it is not essential that the driver be assigned the best possible route. A slightly less optimal route may be necessary to allow routes to be computed in a reasonable amount of time.

 *Heuristic:* an algorithm that approximates a solution, usually in much less time than it would take to find an exact solution.

20.7 Exercises

Solutions to these exercises can be found in Appendix A.18 on page 367.

1. *Problem:* Two algorithms have been created to process input records. The first algorithm processes 100 records in 13 seconds. On the same computer, the second algorithm processes 100 records in only 8 seconds. Which algorithm will perform faster on 200 records?
2. *Problem:* One algorithm was step-counted and found to complete in $4n + n^2 + \log n$ steps (with input size n). A second algorithm was step-counted and found to complete in $2n^3 + 4$ steps (with input size n). Which algorithm has the better time complexity?
3. *Problem:* Find the time complexity of the following algorithm:
 - (a) Input a and b as positive whole numbers
 - (b) Let n be the larger of a and b
 - (c) If $a \div n$ is a whole number, AND
 - (d) If $b \div n$ is a whole number, return n as the answer.
 - (e) Otherwise, modify $n = n - 1$
 - (f) Go to line (b)
4. *Problem:* Find the time complexity of Euclid's Algorithm.
5. *Problem:* Find the time complexity of the following algorithm:
 - (a) Let $item_i$ represent the i^{th} item in a list of n numbers, with $item_0$ being the first element.
 - (b) Let $k = 0$
 - (c) If $k = n$ then return true
 - (d) Let $z = 0$
 - (e) If $z = k$ then go to line (i)
 - (f) If $item_z > item_k$ then return false
 - (g) Update $z = z + 1$
 - (h) Go to line (e)
 - (i) Update $k = k + 1$
 - (j) Go to line (c)
6. *Problem:* Devise an alternative, more efficient, algorithm which computes the same result as the algorithm given in the previous problem. Find the time complexity of the improved algorithm.
7. *Problem:* Your company has a records processing algorithm which runs overnight to process the day's sales. When the algorithm was first implemented, there were about 20 sales a day, and the algorithm took about five minutes to run. A few months later, daily sales averaged about 100 per day, and you notice the algorithm is now taking about two hours to run. The newly hired sales manager claims she can triple the company's sales. If she does, will the algorithm still finish in time for business open at 8:00AM if it is started at 5:00PM when the business closes?



8. *Problem:* Given the previous solution, Mike the IT guy notes that all sales are currently being processed on one server. He proposes buying several more servers (which perform at the same speed as the current server) to distribute the load.

- (a) Assuming the load can be equally distributed between servers, how many servers would be required to complete the job in time?
- (b) If the total sales per day increases to 400, how many servers would be required to complete the job in time?

Chapter 21

Expression Trees

PART VI

Supporting Material

A Solutions	240
A.1 Sets	241
A.2 Counting	244
A.3 Venn Diagrams	248
A.4 Simplifying Set Expressions . .	256
A.5 Logical Operators and Truth Tables	261
A.6 Manipulating Logical Expressions	270
A.7 Decision Tables	277
A.8 Logic Circuits	284
A.9 Number Systems	297
A.10 Integer Numbers	303
A.11 Floating Point Numbers	309
A.12 Unicode and ASCII	314
A.13 Images and Color	317
A.14 Bitwise Operations and Mask- ing	322
A.15 Error Correcting Codes	327
A.16 Digital Logic	339
A.17 Flowcharts	355
A.18 Analysis of Algorithms	367
B Bibliography	373
B.1 Wikipedia	373
B.2 Websites	374
B.3 Books	376
C Glossary	377
.	377

Appendix **A**

Solutions

Fully worked solutions for all exercises are included for each chapter.

Contents

A.1 Sets	241
A.2 Counting	244
A.3 Venn Diagrams	248
A.4 Simplifying Set Expressions	256
A.5 Logical Operators and Truth Tables	261
A.6 Manipulating Logical Expressions	270
A.7 Decision Tables	277
A.8 Logic Circuits	284
A.9 Number Systems	297
A.10 Integer Numbers	303
A.11 Floating Point Numbers	309
A.12 Unicode and ASCII	314
A.13 Images and Color	317
A.14 Bitwise Operations and Masking	322
A.15 Error Correcting Codes	327
A.16 Digital Logic	339
A.17 Flowcharts	355
A.18 Analysis of Algorithms	367

A.1 Sets

Exercises found in Chapter 1 on page 8.

Assume $A = \{1, 2, 3\}$, $B = \{2, 3, 4\}$, $C = \{4, 5, 6\}$, $D = \{1, 3, 5\}$, $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

1. *Problem:* Find the set described by $A \cup (B \cap D')'$.

Solution: Keep order of operations in mind, and solve.

- (a) $A \cup (B \cap D')'$
- (b) $\{1, 2, 3\} \cup (\{2, 3, 4\} \cap \{1, 3, 5\})'$ (insert set definitions)
- (c) $\{1, 2, 3\} \cup (\{2, 3, 4\} \cap \{2, 4, 6, 7, 8, 9, 10\})'$ (parentheses first, and then complement inside)
- (d) $\{1, 2, 3\} \cup (\{2, 4\})'$ (parentheses first)
- (e) $\{1, 2, 3\} \cup \{1, 3, 5, 6, 7, 8, 9, 10\}$ (complement)
- (f) $\{1, 2, 3, 5, 6, 7, 8, 9, 10\}$ (union)

Therefore, $A \cup (B \cap D')' = \{1, 2, 3, 5, 6, 7, 8, 9, 10\}$. Everything in the universe except the 4.

2. *Problem:* Find the set described by $A \cup B \cap D$.

Solution: Keep order of operations in mind. Intersection has a higher precedence than union, so (in the absence of parentheses) complete the intersection first.

- (a) $A \cup B \cap D$
- (b) $\{1, 2, 3\} \cup \{2, 3, 4\} \cap \{1, 3, 5\}$ (insert set definitions)
- (c) $\{1, 2, 3\} \cup \{3\}$ (intersection first)
- (d) $\{1, 2, 3\}$ (then the union)

Therefore $A \cup B \cap D = \{1, 2, 3\}$. Note that attempting to solve using a different order of operations would cause a different result. Always put parentheses around an expression if the order may be unclear.

3. *Problem:* Determine if $A \cap B \subset B \cup D$.

Solution: Keep in mind that proper subset \subset requires that all elements on the left appear on the right, and also that the right must have at least one additional element.

- (a) $A \cap B \subset B \cup D$
- (b) $\{1, 2, 3\} \cap \{2, 3, 4\} \subset \{2, 3, 4\} \cup \{1, 3, 5\}$ (insert set definitions)
- (c) $\{2, 3\} \subset \{1, 2, 3, 4, 5\}$ (perform intersection and union)



- (d) $\{2, 3\} \subset \{1, 2, 3, 4, 5\}$ (check if all elements on the left appear on the right; plus at least one more element on the right)

The two and three appear on the right, and the right side has at least one additional element, so $A \cap B \subset B \cup D$.

4. *Problem:* Find $|A \cap C|$.

Solution: The vertical bars indicate that the cardinality, that is, number of elements in the set, is desired. We must first find the contents of the set, and then count them.

- (a) $|A \cap C|$
- (b) $|\{1, 2, 3\} \cap \{4, 5, 6\}|$ (insert set definitions)
- (c) $|\emptyset|$ (perform intersection)
- (d) 0 (count the number of items in the set - the empty set has no items)

Therefore, $|A \cap C| = 0$.

Assume $M = \{x : x \text{ is a math student}\}$, $C = \{x : x \text{ is a CIS student}\}$, and $V = \{x : x \text{ plays videogames}\}$. Let $U = \{x : x \text{ is a student at the college}\}$. Note that all sets must be subsets of the universe, so in this case the set V implicitly is limited to only those students at the college who play videogames; not videogame players outside the college.

5. *Problem:* Write a set expression which gives the set of all CIS students who play videogames.

Solution: We recognize that CIS students are represented by C , and that videogame players are represented by V . The question asks for the result to be a set, so the standard set operators of union, intersection, and complement are available. There are two main ways to combine these sets: $C \cup V$ or $C \cap V$. The first, union, indicates all CIS students (regardless of whether or not they play videogames) together with all videogame players (regardless of whether or not they are CIS students). This would not be the correct set. The second, intersection, indicates all CIS students who also play videogames, which is what the problem asked for.

Therefore, the set of all CIS students who play videogames is represented by $C \cap V$.

6. *Problem:* Write a set expression which gives the set of all students who play videogames and are either math or CIS students, or both.

Solution: In this case, we can break the problem down into parts. The first part "play videogames" suggests the set V . The second part "either math or CIS students, or both" suggests the use of the union operator with the sets M and C . Finally, the "and" which

connects them together requires that the student be a member of both subparts to be part of the whole, so we'll use the intersection operator.

Therefore, the set of all students who play videogames and are either math or CIS students, or both is represented by $V \cap (M \cup C)$. Because intersection and union have the same precedence, the use of parentheses here is important. If parentheses were not used, the expression would actually describe videogame players who were also math students, together with all CIS students.

7. *Problem:* Write a set statement which indicates that all CIS students play videogames.

Solution: Notice the phrasing of this problem is different. We are no longer looking for a set; we are now looking for a truth claim. This requires the use of equality or subset operators to make a claim about two related sets. It is important to note the directionality of this statement: all CIS students play videogames, but not all videogame players are CIS students (at least, we can't assume so at this time). This means that every member of the set CIS students must also be a member of the set videogame players. This denotes a subset situation.

Therefore, we can indicate that all CIS students play videogames by claiming that $C \subseteq V$.

8. *Problem:* Write a set statement which indicates that some math students don't play videogames.

Solution: The word "some" means one or more, or "there exists". The statement must claim that there is some element that is both in the math students set and in the don't play videogames set.

The math set is represented by M , and the don't play videogames set is V' . Notice the use of ' to indicate complement; that is, all students who DO NOT play videogames. Now we must claim that there is at least one member in common between these sets. If we intersect them, then the result must not be empty.

Therefore, we can indicate that some math students don't play videogames with $M \cap V' \neq \emptyset$.

Alternative Solution: If we consider the set of math students, there must be at least one who is not in the set of videogame players. Thus, it is not possible that math students could be a subset of videogame players. By refuting such a subset, we can enforce the same claim: that some (at least one) math students don't play videogames.

Therefore, we can indicate that some math students don't play videogames with $M \not\subseteq V$.



A.2 Counting

Exercises found in Chapter 2 on page 18.

1. *Problem:* Sam buys a box of mixed chocolates. The box contains 3 nut varieties, 5 truffle varieties, 3 caramels, and 2 hard chocolates. Suzie opens the box first and eats 4 chocolates; but she doesn't like truffles or hard chocolates, so she won't eat any of those. Sam doesn't like nut varieties or hard chocolates. How many chocolates are left that Sam might like?

Solution: Given that Suzie doesn't like (and won't eat) truffles or hard chocolates, that leaves nut varieties and caramels. There are only three caramels in the box, so even if she ate all three she would still have eaten one nut variety. The other extreme is that she ate all three nut varieties and then had just one of the caramels.

Sam will eat only truffles and caramels. Since Suzie doesn't like truffles, we know that all the truffles must remain. That's five.

Of the caramels, Suzie may have eaten at least one and at most all three of the caramels, leaving between zero and two caramels remaining. That leaves a total of five to seven chocolates that Sam might like.

2. *Problem:* Given that $|A| = 5$ and $|B| = 11$, but not knowing the details of the contents, what is $|A \cap B|$?

Solution: In this case, we are given only the cardinality but not the contents of the two sets, and are asked to find how many elements will appear in the intersection. There are two extremes: In one case, $A \subset B$, that is, every element in A appears in B . In that case, $A \cap B = A$. We already know that $|A| = 5$.

In the other extreme, the sets may be disjoint. In that case, $A \cap B = \emptyset$. We know that $|\emptyset| = 0$. The intersection set cardinality must be between zero and five, inclusive.

Therefore, $0 \leq |A \cap B| \leq 5$.

3. *Problem:* Given that $|A| = 4$ and $|B| = 6$, but not knowing the details of the contents, what is $|A \cup B|$?

Solution: Likewise, again, we are given only the cardinality but not the contents of the two sets, and are asked to find how many elements will appear in the union. There are two extremes: In one case, $A \subset B$, that is, every element in A appears in B . In that case, $A \cup B = B$ (because duplicates don't exist in sets). We already know that $|B| = 6$.

In the other extreme, the sets may be disjoint. In that case, the union of A and B will be a set consisting of all elements in A , and then all elements in B . In other words, if A and B are disjoint, then $|A \cup B| = |A| + |B|$. So at most the union could have 10 elements.

Therefore, $6 \leq |A \cup B| \leq 10$.



Alternative Solution: Recall the formula $|A \cup B| = |A| + |B| - |A \cap B|$. In this case, $|A|$ and $|B|$ are known, so we must determine $|A \cap B|$. In a similar way to the previous exercise, the range of an intersection is from zero to the number of elements in the smaller set. In this case $0 \leq |A \cap B| \leq 4$. Starting with $|A| + |B| = 10$, and subtracting the range given to both sides, we find $6 \leq |A \cup B| \leq 10$.

4. *Problem:* An IPv4 network address consists of 32 bits. Each bit has two possibilities (0 or 1). What is the maximum theoretical number of possible IPv4 addresses?

Solution: The two questions here are order and repetition. Both are yes: order is important because the defined address is based on the bit sequence. Like house addresses, the address number 123 would be different from 321. Repetition is also essential, as we are choosing 32 items from a pool of only 2 (each bit has two possible values).

Choosing permutations with repetition gives us the formula n^r , where $n = 2$ and $r = 32$. Therefore $2^{32} = 4294967296$.

Note: in practice, fewer addresses are available due to certain ranges being classified for special or restricted use.

5. *Problem:* There are 6 boxes numbered 1, 2, 3, 4, 5, and 6. Each box is to be filled up either with a red or a green ball in such a way that at least 1 box contains a green ball and the boxes containing green balls are consecutively numbered. How many different arrangements are possible?

Solution: A reasonable approach to this problem is to examine what kind of assumptions we can make. For example, we know that at least one green ball exists, and that all the green balls are together. Therefore, we can represent the green balls as a starting position and a length. All other balls will be red.

The valid lengths are 1 through 6 (one being the minimum number of green balls, and six meaning every spot has a green ball). The valid starting positions vary based on the length: When the length is 1, then any of the six boxes are valid starting positions. When the length is 2, only boxes one through five are valid starting positions. Finally, when the length is 6, only box 1 is valid for starting.

In this way, we can use the additive counting rule to combine these options, since exactly one is chosen. For each length 1 through 6, we'll add the possible number of starting positions, giving us $6 + 5 + 4 + 3 + 2 + 1 = 21$.

6. *Problem:* How many different four letter words can be formed (the words need not be meaningful) using the letters of the word MEDITERRANEAN such that the first letter is E and the last letter is R?

Solution: One key assumption that we must make is whether or not we are allowed to use a letter more times than it appears in the original word. For example, is ETTR a valid



answer? (Note there is only one T in the original word). For this exercise, we'll assume that letters are not usable more times than they appear in the original.

Given that the new word must start with E and end with R, we really have two spaces to fill in, and we must take away an E and an R from the original word, giving us MDITER-ANEAN. (I removed the first E and the first R, although any single of each is possible).

We now have eleven letters remaining, and two spots to fill. Repetition is not allowed (because each letter from the original word can only be used once) and order is important. That gives us permutations without repetition. However, if we were to choose two of the same letter (possible only with E, A, or N), then that should be counted as one possibility, but a naive permutation approach will count them as two possibilities (e.g. EAAR with the first A and then the second A, and EAAR with the second A and then the first A).

To solve this problem, we will count the duplicate letter case separately, and reduce our original word to MDITERAN. We can now use the permutation without repetition formula,

$$\text{finding that } \frac{n!}{(n-r)!} = \frac{8!}{(8-2)!} = \frac{8!}{6!} = 8 * 7 = 56.$$

We need to add in the duplicate letter cases. There are three: EAAR, EEER, and ENNR. Therefore, we find a total of 59 possible arrangements.

7. *Problem:* How many ways can a class of six people be split into two equal groups if there is no distinction between the groups?

Solution: In this case, there is no labeling of the groups (e.g. group A or group B), so the partition formula won't work as-is. The reason for this is that if we have the groups $A = \{1, 2, 3\}$ and $B = \{4, 5, 6\}$; compared to $A = \{4, 5, 6\}$ and $B = \{1, 2, 3\}$, these are the same two groups, but the partition formula would consider them different.

However, notice that every group selection has a "mirror image". So we could use the partition formula and then divide the number of results in half to eliminate the mirror image.

The partition formula is $\frac{n!}{r_1! * r_2! * r_3! * \dots}$. In this case, $n = 6$, $r_1 = 3$, and $r_2 = 3$. This meets the requirement that $n = r_1 + r_2$.

$$\frac{6!}{3! * 3!} = \frac{6 * 5 * 4}{3!} = 20. \text{ Cutting the result in half gives us 10 possible arrangements.}$$

Alternative Solution: We can select, without ordering or repetition, three members of the class to form one group using the combinations without repetition formula. Again, we must divide the total number of possibilities in half to counter the effect where students $\{1, 2, 3\}$ are selected and $\{4, 5, 6\}$ are not; this is the same as $\{4, 5, 6\}$ being selected and $\{1, 2, 3\}$ not.

Combinations without repetition uses the binomial coefficient formula $\binom{n}{r}$.



$\binom{n}{r} = \binom{6}{3} = \frac{6!}{3! * (6 - 3)!} = \frac{6!}{3! * 3!} = \frac{6 * 5 * 4}{3!} = 20$. Cutting the result in half gives us 10 possible arrangements.

8. *Problem:* A restaurant offers the following menu: select either an entree or a burger. There are three entrees available, and each entree comes with two sides. There are four sides to choose from. We can choose two of the same side, if desired. If we choose the burger, on the other hand, there are five different burgers available, but burgers do not come with any sides. How many different selections are possible?

Solution: Divide the problem into two parts: the entree part and the burger part. Because we are choosing one from this group, we will combine them with the additive counting rule. The entree part consists of one of three entrees and two of four sides.

Two of four sides is the interesting part: in this case, we have been told that repetition is allowed, and we can assume that order does not matter (since the sides come together).

Therefore, we choose combinations with repetition, represented by $\binom{n + r - 1}{r}$. In this case, $n = 4$ and $r = 2$. Therefore, $\binom{4 + 2 - 1}{2} = \binom{5}{2} = \frac{5!}{2! * (5 - 2)!} = \frac{5!}{2! * 3!} = \frac{5 * 4 * 3!}{2! * 3!} = \frac{5 * 4}{2!} = \frac{20}{2} = 10$.

There are ten different ways of selecting the sides. Working backwards, there are three entrees, and we are selecting one, so there are just three possible choices. Because we choose one entree and one of the ten ways of choosing sides, there are a total of $3 * 10 = 30$ ways of choosing an entree and sides. If we choose a burger instead, there are five burgers.

So the total number of possible menu choices is 35.

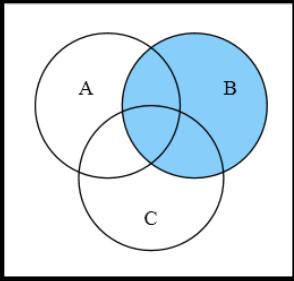
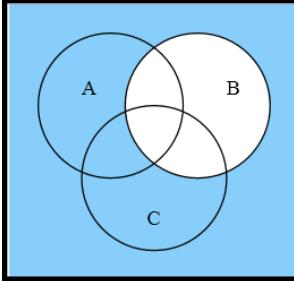
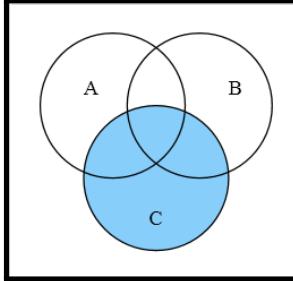
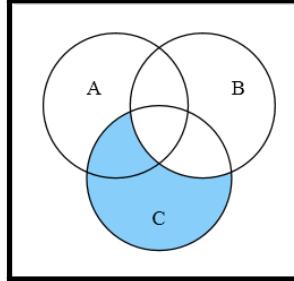
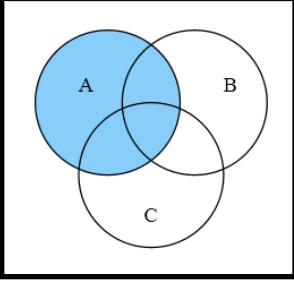
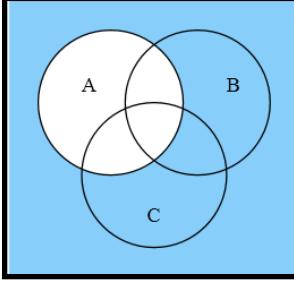
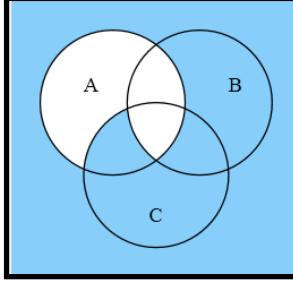


A.3 Venn Diagrams

Exercises found in Chapter 3 on page 26.

1. *Problem:* Draw a Venn diagram to visualize $A' \cup B' \cap C$.

Solution: In the usual order of operations, we will first investigate the intersection to solve $B' \cap C$.

First, find the inner-most set B :	The complement is the next step, so fill in the opposite of everything that was filled in before:	Separately, find the set C :	Intersect the previous two sets to find $B' \cap C$:
			
Solving the left portion, find the set A :	The complement will be everything except A :	Now union the A' set with the previous $B' \cap C$ set for the final answer:	
			

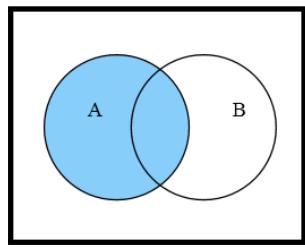
2. *Problem:* Determine if $A' \cap B' = (A \cup B)'$.

Solution: Equality of set expressions can be determined by drawing a Venn diagram for each set expression in question. If the two Venn diagrams are the same, then the set expressions are equal; otherwise they are not equal.

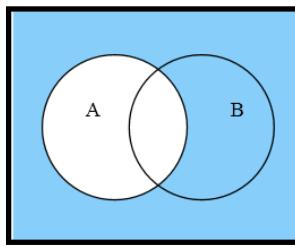
First, we will find the Venn diagram for $A' \cap B'$.



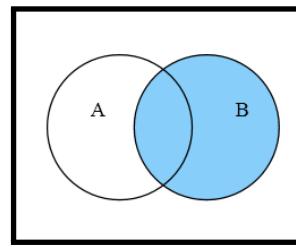
First, find the innermost set A :



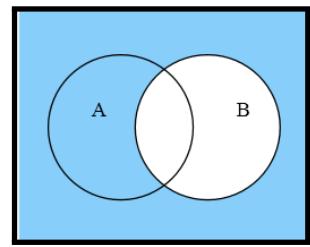
The complement is next step, so fill in the opposite of everything that was filled in before:



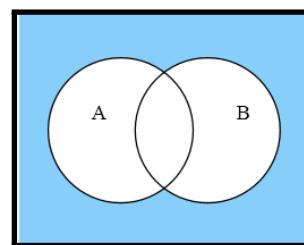
Separately, find the set B :



Likewise, find the complement of B :

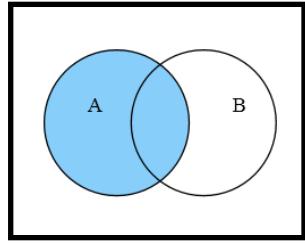


We can intersect the two results, A' and B' , to find all areas that exclude both A and B :

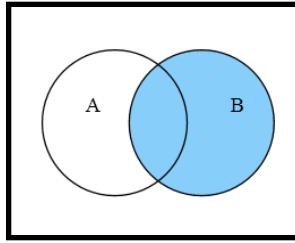


Now, let's find the Venn diagram for $(A \cup B)'$ and see if it is the same.

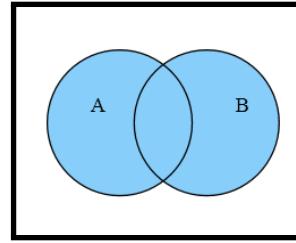
First, find the innermost set A :



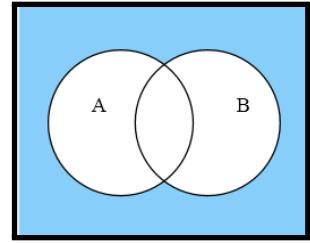
Separately, find the set B :



Next, union the two sets: B :



Finally, invert the last diagram to apply the complement:



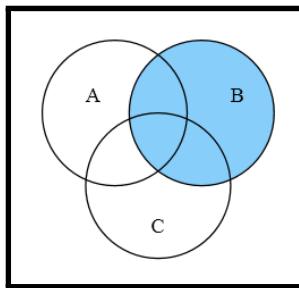
Note that both final diagrams are the same. Therefore, the two expressions $A' \cap B'$ and $(A \cup B)'$ are equal. This equivalence is called DeMorgan's Law.

Let the universe be customers of a bank. Let the set $A = \{x : x \text{ has a savings account}\}$, $B = \{x : x \text{ is a preferred customer}\}$, and $C = \{x : x \text{ has a checking account}\}$.

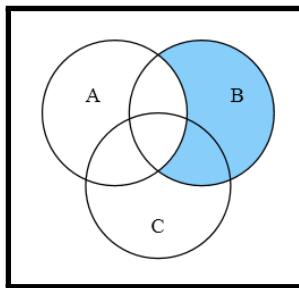
3. *Problem:* Draw a Venn diagram illustrating preferred customers who don't have a savings account.



Solution: First, start with all preferred customers. Note the overlap with A is those customers who are preferred and also have a savings account. The overlap with C is those customers who are preferred and also have a checking account. The triangle area in the middle is preferred customers who have both a checking and savings account.



The problem asks for those preferred customers who do not have a savings account, so we must reduce this diagram by excluding the savings account holders. Both the top overlap with A and the middle triangle include savings account holders, so we will remove those from the diagram:

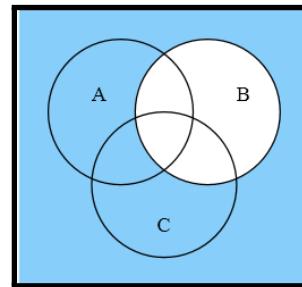


Alternative Solution: Convert the English sentence into a set expression. We find preferred customers B and those who don't have a savings account as A' . The problem wants both preferred AND no savings account, which gives us $B \cap A'$.

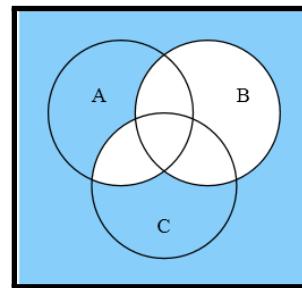
First, find the innermost set B :	Separately, find the set A :	Find the complement of A :	Combine B and A' with an intersection:

4. *Problem:* Draw a Venn diagram illustrating non-preferred customers, except those who hold both a checking and savings account.

Solution: First, start with the customers who are not preferred. These are customers outside of the preferred set B :



The next phrase starts with “except”, which means we will be further reducing the set as shown above. We want to remove those who hold both a checking and savings account. This is a simple matter of removing the wedge where A and C overlap, as this indicates the customers who have both types of accounts:

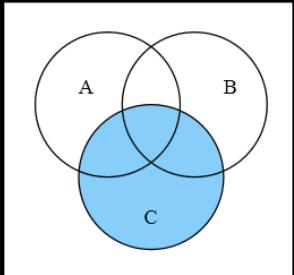
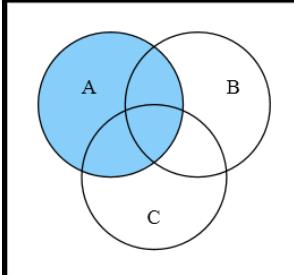
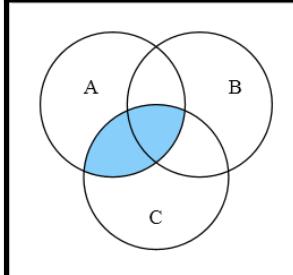
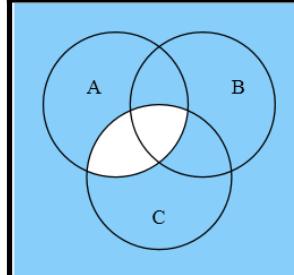


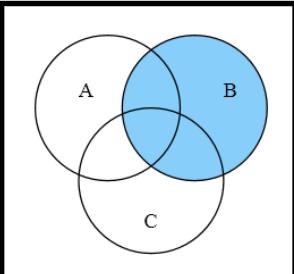
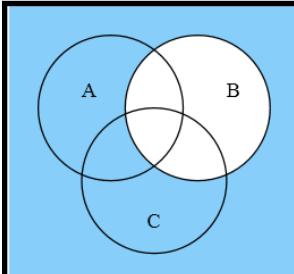
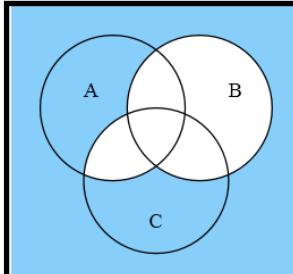
Alternative Solution: Convert the English sentence into a set expression. First, we will transform the English slightly to identify the two main parts: non-preferred customers, and customers who hold both a checking and savings account. The non-preferred customers are B' , and the customers with both accounts can be represented by an intersection: $C \cap A$.

These are combined with “except”, which indicates set difference (that is, non-preferred customers minus customers who hold both a checking and savings account). So our expression at this point is $B' \setminus (C \cap A)$.

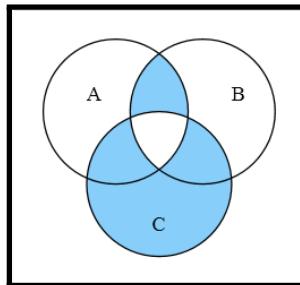
Recalling that set difference is defined as $A \setminus B = B' \cap A$. Plugging our sentence above into this definition gives a final set expression of $(C \cap A)' \cap B'$. We can now draw this Venn diagram in the usual way:



First, find the inner-most set C :	Separately, find the set A :	The intersection of these two is $C \cap A$:	Take the complement of this result to find $(C \cap A)'$:
			

Now, find the set B :	The complement of this set is B' :	Combine B' with $(C \cap A')$ from the previous line using intersection:	
			

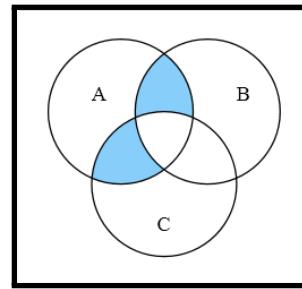
5. *Problem:* Determine a set expression that matches the following Venn diagram:



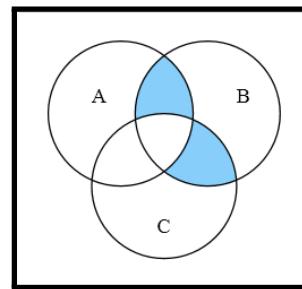
Solution: A reasonable way to approach this type of problem is to consider which parts of each set, if any, are included in the diagram.

For the set A , we find that only the parts where A overlaps with B , or where A overlaps with C , are included. Notice the central triangle is excluded, so when B is included, C must be excluded; likewise, when C is included, B must be excluded. We can write that portion as a set expression: $(A \cap B \cap C') \cup (A \cap C \cap B')$.

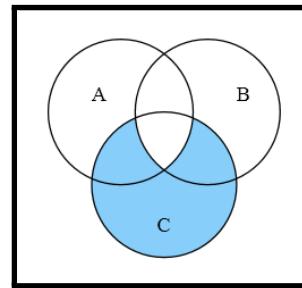




For the set B , we likewise find that most of B is not present; instead, only the parts where B overlaps A or C , but not both, are included. We can render this part with the expression: $(B \cap A \cap C') \cup (B \cap C \cap A')$



Finally, for the set C , we find everything is included except for the intersection of all three sets, so we can exclude the middle with the expression $C \cap (A \cap B)'$.



To complete the expression, we can union all these parts together, omitting any duplicate sub-expressions. This gives us the set expression $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (B \cap C \cap A') \cup (C \cap (A \cap B)')$. This expression is substantially larger than necessary, and can be reduced either through general reasoning, or through specific transformation rules which will be discussed in detail in the next chapter.

Alternative Solution: A closer look at the original diagram reveals two distinct parts: the C part and the middle $A \cap B$ wedge. A key insight is to realize that the relationship between C and $A \cap B$ in the diagram is that of symmetric difference (one or the other, but not both). In other words, the diagram can be represented by the set expression $C \Delta (A \cap B)$.

Recall the definition of symmetric difference given previously, $A \Delta B = (A \cup B) \cap (A \cap B)'$. We can insert C and $(A \cap B)$ into the definition, coming up with a final set expression:

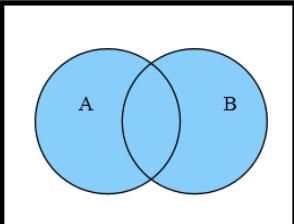
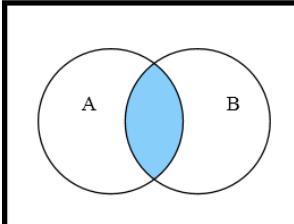
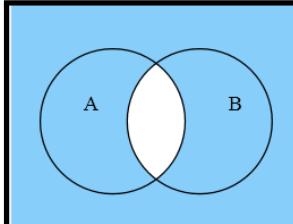
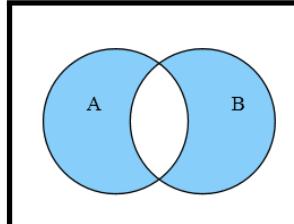


$(C \cup (A \cap B)) \cap (C \cap (A \cap B))'$. Although this set expression is different from the set expression in the previous solution, they are both correct because they both visualize the same Venn diagram.

6. *Problem:* Symmetric difference is defined earlier in this chapter as $A \Delta B = (A \cup B) \cap (A \cap B)'$. An alternative definition of symmetric difference is given by $A \Delta B = (A \setminus B) \cup (B \setminus A)$. Prove that these two definitions are equivalent.

Solution: Begin by substituting in the definition for set difference into the second formula in order to find a standard set expression. Recall the definition of set difference: $A \setminus B = B' \cap A$. Therefore, the second definition expands to $A \Delta B = (B' \cap A) \cup (A' \cap B)$. We now must show that two set expressions are equivalent. The two expressions are $(A \cup B) \cap (A \cap B)'$ and $(B' \cap A) \cup (A' \cap B)$. A quick check for equivalence at this point is to see if the two expressions are identical. If so, they are trivially equivalent. However, the two expressions are not identical. Therefore, we must move on to a more sophisticated check. If these two expressions are equivalent, then they must have the same Venn diagram.

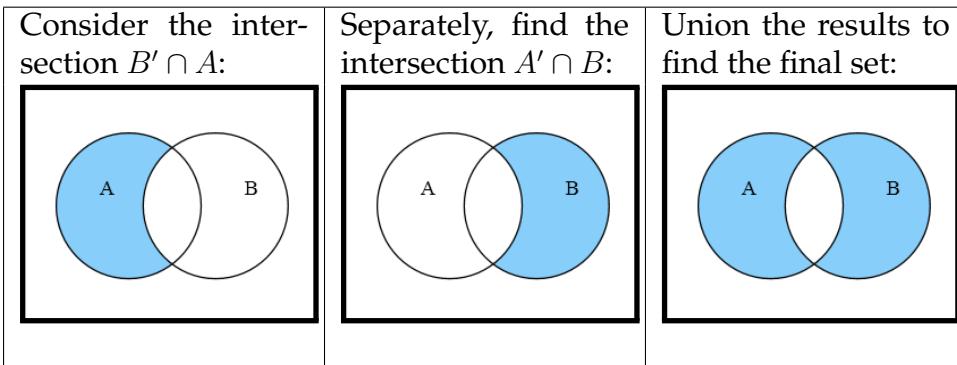
First, create the Venn diagram for $(A \cup B) \cap (A \cap B)'$.

Consider the union $A \cup B$:	Separately, find the intersection $A \cap B$:	Take the complement of the intersection to find $(A \cap B)'$:	Intersect the two results to find the final set:
			

As a sanity check, confirm that the final Venn diagram does in fact reflect the concept of symmetric difference: one or the other, but not both.

Second, create the Venn diagram for $(B' \cap A) \cup (A' \cap B)$. Be careful to note that $B' \cap A$ and $A' \cap B$, although lexically similar, are not the same. For conciseness the construction of $B' \cap A$ and $A' \cap B$ are omitted, although these could be constructed following the same techniques shown previously.





The final diagram in both cases are the same, therefore, the two set expressions are proven equivalent. Both definitions of symmetric difference are correct.

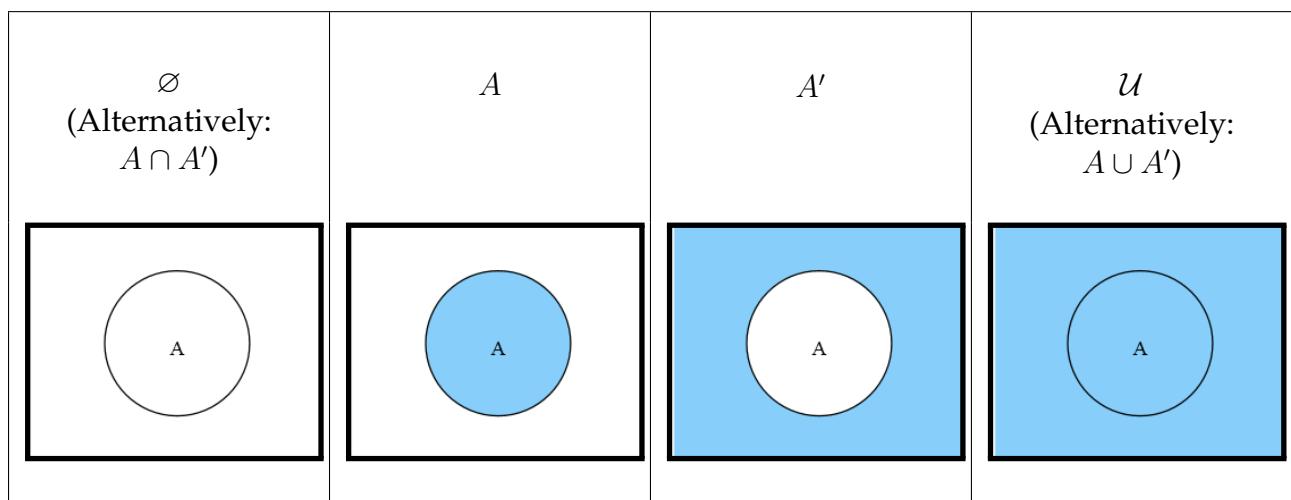
7. *Problem:* How many different Venn diagrams of two sets are possible?

Solution: Recall that a Venn diagram of n sets consists of 2^n regions. In this case, a Venn diagram of two sets has $2^2 = 4$ regions (namely, outside both sets, just the A part, just the B part, and the intersection of the two). Each region is either selected or not selected ($n = 2$) and there are four regions ($r = 4$). Order is important, because each region is specific (choosing two regions isn't enough information; we must know which two are chosen). Repetition is allowed, because more than one region can be selected. Therefore, we have permutations with repetition, counted with the formula n^r .

Plugging in our values for n and r , we get the formula $2^4 = 16$. There are (only) sixteen different Venn diagrams of two sets.

8. *Problem:* There are four possible Venn diagrams of one set. Enumerate set expressions for each.

Solution: The four possibilities are: nothing (the empty set), just A , everything except A , and everything (the universe).

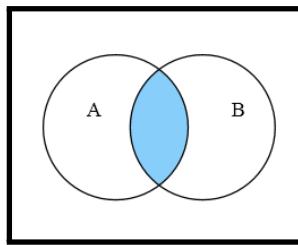


A.4 Simplifying Set Expressions

Exercises found in Chapter 4 on page 35.

1. *Problem:* Simplify the set expression $(A' \cup B')'$.

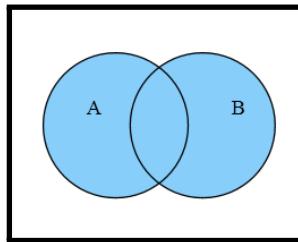
Solution: First, create the Venn diagram for the expression so we can verify when we have reached the final simplified expression. We see from the diagram that the final simplified expression will be $A \cap B$.



1. $(A' \cup B')'$ Initial Set
2. $(A' \cup B')'$ DeMorgan's Law 8a
3. $A'' \cap B''$ Double Negation Law 6 (applied in both places)
4. $A \cap B$ Final Expression

2. *Problem:* An incorrect attempt at simplifying $B \cup B' \cap A$ was shown earlier in the chapter. Show a correct simplification for this expression.

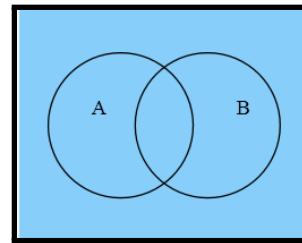
Solution: As seen previously, $B \cup B' \cap A$ will reduce to $A \cup B$.



1. $B \cup (B' \cap A)$ Initial Set with parentheses; Distributive Law 4a
2. $(B \cup B') \cap (B \cup A)$ Complement Law 7a
3. $\mathcal{U} \cap (B \cup A)$ Commutative Law 3b
4. $(B \cup A) \cap \mathcal{U}$ Identity Law 5b
5. $B \cup A$ Final Expression

3. *Problem:* Simplify the set expression $A \cap B \cap A' \cup A \cup B' \cup A'$.

Solution: First, find the Venn diagram for this expression. You'll notice the entire area is filled in, meaning the expression must reduce to the universe, \mathcal{U} .

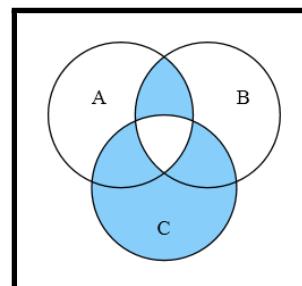


Caution! We cannot grab the $A' \cup A$ out of the middle and start there! With a mix of intersection and union we must be mindful of order of operations.

1. $A \cap B \cap A' \cup A \cup B' \cup A'$ Initial Set
2. $A \cap B \cap A'$ $\cup A \cup B' \cup A'$ Commutative Law 3b
3. $A \cap A'$ $\cap B \cup A \cup B' \cup A'$ Complement Law 7b
4. $\emptyset \cap B$ $\cup A \cup B' \cup A'$ Commutative Law 3b
5. $\emptyset \cap \emptyset$ $\cup A \cup B' \cup A'$ Identity Law 5d
6. $\emptyset \cup A$ $\cup B' \cup A'$ Commutative Law 3a
7. $A \cup \emptyset$ $\cup B' \cup A'$ Identity Law 5a
8. $A \cup B'$ $\cup A'$ Commutative Law 3a
9. $A \cup A'$ $\cup B'$ Complement Law 7a
10. $\mathcal{U} \cup B'$ Commutative Law 3a
11. $B' \cup \mathcal{U}$ Identity Law 5c
12. \mathcal{U} Final Expression

4. **Problem:** In the previous chapter, a certain Venn diagram was found to have the set expression $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (B \cap C \cap A') \cup (C \cap (A \cap B)')$. Simplify this expression.

Solution: Recall the Venn diagram for this expression. An alternative solution to the problem gave us the expression $(C \cup (A \cap B)) \cap (C \cap (A \cap B)')$. However, we can't be sure this is the simplest form.



Due to the size and length of this expression, not all applications of the commutative and associative law will be specified. However, it is important to be sure that all manipulation of order and parentheses obeys the laws and order of operations.



- | | |
|--|---|
| <ol style="list-style-type: none"> 1. $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (B \cap C \cap A') \cup (C \cap (A \cap B)')$ 2. $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (B \cap C \cap A') \cup (C \cap (A \cap B)')$ 3. $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (B \cap C \cap A') \cup (C \cap (A' \cup B'))$ 4. $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (B \cap C \cap A') \cup (C \cap A') \cup (C \cap B')$ 5. $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (C \cap A') \cup (C \cap B')$ 6. $(A \cap B \cap C') \cup (A \cap C \cap B') \cup (C \cap B') \cup (C \cap A')$ 7. $(A \cap B \cap C') \cup (C \cap B') \cup (C \cap A')$ 8. $(A \cap B \cap C') \cup (C \cap (B' \cup A'))$ 9. $(A \cap B \cap C') \cup (C \cap (B \cap A)')$ | Initial Set
DeMorgan's Law 8b
Distributive Law 4b
Absorption Law 9a
Commutative Law 3a
Absorption Law 9a
Distributive Law 4b
DeMorgan's Law 8b
Final Expression |
|--|---|

At this point, there are no clear opportunities for further simplification. So we believe (but cannot prove) that this is likely the simplest form of this expression.

5. *Problem:* Prove that $(A \cap B') \cup (A \cap B) = A \cup (B \cap B')$.

Solution: Equivalence of two expressions can be proven by showing a sequence of law applications starting at one of the expressions and ending at the other. Either direction is acceptable (in other words, you can start with the expression on the left and end with the expression on the right, or vice versa).

1. $(A \cap B') \cup (A \cap B)$ Initial Set
2. $(A \cap B') \cup (A \cap B)$ Distributive Law 4b
3. $A \cap (B' \cup B)$ Commutative Law 3a
4. $A \cup (B \cap B')$ Final Expression

By showing a sequence of law applications which translates one expression into the other, we have proven the two expressions are equivalent. Note that it is not necessary to pass through the simplest form of either expression to prove equivalence.

Alternative Solution: Simplify both expressions. Begin with $(A \cap B') \cup (A \cap B)$.

1. $(A \cap B') \cup (A \cap B)$ Initial Set
2. $(A \cap B') \cup (A \cap B)$ Distributive Law 4b
3. $A \cap (B' \cup B)$ Commutative Law 3a
4. $A \cap U$ Identity Law 5b
5. A Final Expression

Likewise, simplify $A \cup (B \cap B')$.

1. $A \cup (B \cap B')$ Initial Expression
2. $A \cup (B \cap B')$ Complement Law 7b
3. $A \cup \emptyset$ Identity Law 5a
4. A Final Expression

Both expressions simplify to the same expression, thus they must be equivalent.

6. *Problem:* Prove the absorption law (9a) $x \cup (x \cap y) = x$.



Solution: In this case, since we are trying to prove a particular group of laws, we must use only the other laws to go from one expression to the other. Therefore, we can use any laws except the absorption laws 9a and 9b.

1. $x \cup (x \cap y)$ Initial Set
2. $x \cup (x \cap y)$ Identity Law 5b
3. $(x \cap U) \cup (x \cap y)$ Distributive Law 4b
4. $x \cap (U \cup y)$ Identity Law 5c
5. $x \cap U$ Identity Law 5b
6. x Final Expression

We have proven using the set algebra laws that $x \cup (x \cap y) = x$, so the absorption law (9a) holds.

7. *Problem:* Prove the absorption law (9b) $x \cap (x \cup y) = x$.

Solution: If we accept the previous solution which proves $x \cup (x \cap y) = x$, then it is possible to apply the dual law which states that for any algebraic law of sets, if the unions are replaced with intersections and vice versa, and the universe and empty set likewise switched, that modified equivalence will also be true.

By replacing all unions with intersections and all intersections with unions, we get $x \cap (x \cup y) = x$, which is the law we were asked to prove. Therefore, based on the proof for the absorption law (9a) and the dual property, the absorption law (9b) holds.

Alternative Solution: Similar to the previous sequence of steps.

1. $x \cap (x \cup y)$ Initial Set
2. $x \cap (x \cup y)$ Identity Law 5a
3. $(x \cup \emptyset) \cap (x \cup y)$ Distributive Law 4a
4. $x \cup (\emptyset \cap y)$ Identity Law 5d
5. $x \cup \emptyset$ Identity Law 5a
6. x Final Expression

We have proven using the set algebra laws that $x \cap (x \cup y) = x$, so the absorption law (9b) holds.

8. *Problem:* Megan is writing a video game which needs to process space ships with various weapons. Let the universe be all space ships in play, the set L be all space ships with lasers, the set M be all space ships with missiles, and the set D be all space ships with death rays. Any space ships may be configured with any or all weapons (or no weapons).

Megan needs to write a program to find all space ships which meet one or more of the following criteria:

- (a) The ship has both missiles and lasers.
- (b) The ship has death rays or lasers, but not both.
- (c) The ship has missiles.



- (d) The ship has lasers, missiles, and death rays.

Devise a single simplified set expression to find the ships Megan is looking for.

Solution: The first step in solving this problem is to write set expressions for each item.

- (a) The ship has both missiles and lasers. This will be expressed with intersection because we only want the “both” case. $M \cap L$
- (b) The ship has death rays or lasers, but not both. The symmetric difference gives us one or the other, but not both. $D \Delta L$
- (c) The ship has missiles. Just M
- (d) The ship has lasers, missiles, and death rays. Intersection of all three: $L \cap M \cap D$

The original problem called for the ship to meet one or more of the criteria, so if we connect all the criteria with union operators, that will give us the set that meets one or more of the criteria. This gives us an initial set expression of $(M \cap L) \cup (D \Delta L) \cup M \cup (L \cap M \cap D)$.

First, apply the definition of symmetric difference to give $(M \cap L) \cup ((D \cup L) \cap (D \cap L)') \cup M \cup (L \cap M \cap D)$.

1. $(M \cap L) \cup ((D \cup L) \cap (D \cap L)') \cup M \cup (L \cap M \cap D)$ Initial Set
2. $(M \cap L) \cup ((D \cup L) \cap (D \cap L)') \cup \underline{M} \cup (L \cap M \cap D)$ Commutative Law 3a
3. $\underline{(M \cap L)} \cup M \cup ((D \cup L) \cap (D \cap L)') \cup (L \cap M \cap D)$ Absorption Law 9a
4. $\underline{M} \cup ((D \cup L) \cap (D \cap L)') \cup (L \cap M \cap D)$ Commutative Law 3a
5. $M \cup (L \cap M \cap D) \cup ((D \cup L) \cap (D \cap L)')$ Absorption Law 9a
6. $M \cup ((D \cup L) \cap (D \cap L)')$ Final Expression

The D and L bit looks like it could be simplified, but this is already the simplest form to represent symmetric difference unless there were other uses that we could possibly rely on; but there are not.



A.5 Logical Operators and Truth Tables

Exercises found in Chapter 5 on page 47.

1. *Problem:* Let r be true if the roses are red, let d be true if the daffodils are in bloom, and let c be true if the cucumbers are ripe. Write Boolean expressions for the following claims:

- (a) The roses are red and the daffodils are in bloom.
 $r \wedge d$
- (b) Either the cucumbers are ripe, or the daffodils are in bloom, but not both.
 $c \oplus d$, which could also be written as $(c \vee d) \wedge \neg(c \wedge d)$
- (c) Either the roses are red, or the daffodils are in bloom, or both.
Here, the inclusive OR is acceptable. $r \vee d$
- (d) The roses are not red, nor are the daffodils in bloom, but at least the cucumbers are ripe.
The “at least” here has no specific meaning, other than to say that what follows is true, whereas the preceding values are false. $\neg r \wedge \neg d \wedge c$
- (e) If the cucumbers are ripe, then either the roses must be red or the daffodils must be in bloom, or both.
The if statement makes no claim in the case that the cucumbers are not ripe. We can use implication to describe this situation. $c \rightarrow (r \vee d)$, which could also be written as $\neg c \vee (r \vee d)$.

Solution: The main difficulty with translating English sentences into Boolean expressions is handling the “or”, which can mean either inclusive or exclusive OR. In these cases, the nature of the “or” has been clarified. In general cases, additional thinking or clarification may be needed to ensure the correct kind of OR is used.

- (a) The roses are red and the daffodils are in bloom.
 $r \wedge d$
 - (b) Either the cucumbers are ripe, or the daffodils are in bloom, but not both.
The exclusion of both requires the use of exclusive OR. $c \oplus d$, which could also be written as $(c \vee d) \wedge \neg(c \wedge d)$
 - (c) Either the roses are red, or the daffodils are in bloom, or both.
Here, the inclusive OR is acceptable. $r \vee d$
 - (d) The roses are not red, nor are the daffodils in bloom, but at least the cucumbers are ripe.
The “at least” here has no specific meaning, other than to say that what follows is true, whereas the preceding values are false. $\neg r \wedge \neg d \wedge c$
 - (e) If the cucumbers are ripe, then either the roses must be red or the daffodils must be in bloom, or both.
The if statement makes no claim in the case that the cucumbers are not ripe. We can use implication to describe this situation. $c \rightarrow (r \vee d)$, which could also be written as $\neg c \vee (r \vee d)$.
2. *Problem:* Create a truth table for the Boolean expression $(a \wedge b) \vee \neg(a \vee c)$.
- Solution:* First, create the shell of the truth table. There are three variables present here: a , b , and c . Therefore, we will have four columns and eight rows.



a	b	c	$(a \wedge b) \vee \neg(a \vee c)$
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

Next, for each row, plug in the values and solve. For example, the first row gives the expression $(T \wedge T) \vee \neg(T \vee T)$. This expression can be solved stepwise:

$$\begin{array}{ccc}
 (a & \wedge & b) & \vee & \neg & (a & \vee & c) \\
 (T & & T) & & & (T & & T) \\
 \textcolor{orange}{(T)} & T & \textcolor{orange}{T}) & & & \textcolor{orange}{(T)} & T & T) \\
 \textcolor{orange}{(T} & T & \textcolor{orange}{T}) & & F & \textcolor{orange}{(T} & \textcolor{orange}{T} & T) \\
 \textcolor{orange}{(T} & \textcolor{orange}{T} & \textcolor{orange}{T}) & T & F & \textcolor{orange}{(T} & \textcolor{orange}{T} & T)
 \end{array}$$

The outcome for the first row is true. For the second row we have $a = T, b = T$ and $c = F$, giving the expression $(T \wedge T) \vee \neg(T \vee F)$.

$$\begin{array}{ccc}
 (a & \wedge & b) & \vee & \neg & (a & \vee & c) \\
 (T & & T) & & & (T & & F) \\
 \textcolor{orange}{(T} & T & \textcolor{orange}{T}) & & & \textcolor{orange}{(T} & T & F) \\
 \textcolor{orange}{(T} & T & \textcolor{orange}{T}) & & F & \textcolor{orange}{(T} & \textcolor{orange}{T} & F) \\
 \textcolor{orange}{(T} & \textcolor{orange}{T} & \textcolor{orange}{T}) & T & F & \textcolor{orange}{(T} & \textcolor{orange}{T} & F)
 \end{array}$$

The third row gives us $a = T, b = F$ and $c = T$. Therefore the expression is $(T \wedge F) \vee \neg(T \vee T)$.

$$\begin{array}{ccc}
 (a & \wedge & b) & \vee & \neg & (a & \vee & c) \\
 (T & & F) & & & (T & & T) \\
 \textcolor{orange}{(T} & F & \textcolor{orange}{F}) & & & \textcolor{orange}{(T} & T & T) \\
 \textcolor{orange}{(T} & F & \textcolor{orange}{F}) & & F & \textcolor{orange}{(T} & \textcolor{orange}{T} & T) \\
 \textcolor{orange}{(T} & \textcolor{orange}{F} & \textcolor{orange}{F}) & F & F & \textcolor{orange}{(T} & \textcolor{orange}{T} & T)
 \end{array}$$

This process is repeated for every row.



a	b	c	$(a \wedge b) \vee \neg(a \vee c)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

Alternative Solution:

The expression $(a \wedge b) \vee \neg(a \vee c)$ can be broken down from the top first. We see the top most operator is an OR, which means if either or both terms are true, the whole expression is. Applying short circuit evaluation, we first fill in as true any rows where $a \wedge b$ is true.

a	b	c	$(a \wedge b) \vee \neg(a \vee c)$
T	T	T	T
T	T	F	T
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

Next, we consider the right hand side of $\neg(a \vee c)$. In this case, we are looking for any rows not already claimed where both a and c are *false*. The reason for this is because of the NOT out front – which turns the expression into “neither a nor c ”, the same as “both a and c are false”. There are only two such rows, the bottom, and another near the bottom. These become true as well.

a	b	c	$(a \wedge b) \vee \neg(a \vee c)$
T	T	T	T
T	T	F	T
T	F	T	
T	F	F	
F	T	T	
F	T	F	T
F	F	T	
F	F	F	T

Finally, any remaining unclaimed rows must be false.



a	b	c	$(a \wedge b) \vee \neg(a \vee c)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

3. *Problem:* Create a truth table for the Boolean expression $a \wedge (T \vee b) \wedge (c \wedge F)$.

Solution: Again, check how many distinct variables are present. There are three: a , b , and c . The values T and F are constants, not variables, so these do not affect the size of the truth table.

First, create the empty table:

a	b	c	$a \wedge (T \vee b) \wedge (c \wedge F)$
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

The best way to approach this problem is to use the short-circuiting techniques. We notice that the expression starts with $a \wedge \dots$ meaning that if a is false, the entire expression must be false.

a	b	c	$a \wedge (T \vee b) \wedge (c \wedge F)$
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F



That's half the rows just like that. The next subexpression is $(T \vee b)$. This subexpression will always be true, because true OR anything will be true. We're in an AND clause here, so this subexpression is essentially meaningless. Moving along, the next AND clause is $(c \wedge F)$. Is it possible for a term AND with false to ever be true? No, this subexpression will always be false. And short-circuiting tells us that anything AND false is always false, so...

a	b	c	$a \wedge (T \vee b) \wedge (c \wedge F)$
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

This Boolean expression is always false, so it's probably not a very useful expression.

4. *Problem:* Is $\neg a \wedge \neg b$ equivalent to $\neg(a \vee b)$?

Solution: The only surefire way to test for equivalence of two Boolean expressions is to create a truth table for each, and compare them. In both cases, we have two variables a and b , so the framework of the truth table can be constructed:

a	b	$\neg a \wedge \neg b$
T	T	
T	F	
F	T	
F	F	

To complete the first truth table, insert the various values per row into the expression. Recall that NOT is applied first, and then the AND. For example, the first entry becomes $\neg T \wedge \neg T$ which, applying the NOTs becomes $F \wedge F$, which is false. The second line $\neg T \wedge \neg F$ which, applying the NOTs becomes $F \wedge T$, which is also false. The only true outcome occurs when both inputs are false.

a	b	$\neg a \wedge \neg b$
T	T	F
T	F	F
F	T	F
F	F	T



In the case of $\neg(a \vee b)$, we now have parentheses to deal with, so apply those first. The subexpression $a \vee b$ will be true whenever either or both are true, so that is every row except the last. The NOT operator inverts this, so the expression will be false on every row except the last.

a	b	$\neg(a \vee b)$
T	T	F
T	F	F
F	T	F
F	F	T

The truth tables are the same, so the two expressions are equivalent.

Alternative Solution: Translate the first expression into set notation: $\neg a \wedge \neg b$ becomes $A' \cap B'$. Apply DeMorgan's Law (8a), giving $(A \cup B)'$. Translate this back into Boolean notation: $\neg(a \vee b)$. This is the exact second expression, so they must be equivalent.

In general, all of the set expression simplification laws also apply to Boolean expressions and can be used directly, as will be shown in the next chapter.

5. *Problem:* Prove that $a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$.

Solution: First, recall the truth table for $a \rightarrow b$.

a	b	$a \rightarrow b$
T	T	T
T	F	F
F	T	T
F	F	T

We can “plug in” the opposite values to get the truth table for $b \rightarrow a$. Be very careful to note that the column ordering for the input values is now backwards! We could have also rearranged the rows to have the usual ordering.

b	a	$b \rightarrow a$
T	T	T
T	F	F
F	T	T
F	F	T

These two expressions are being combined with AND. Therefore, the final expression will only be true when both subexpressions are true. We'll construct a new truth table and fill in the result row by row, comparing each pair of matching rows in the above tables. Be sure to match the rows by values and not position!



a	b	$(a \rightarrow b) \wedge (b \rightarrow a)$
T	T	T
T	F	F
F	T	F
F	F	T

Compare this to the truth table for $a \leftrightarrow b$. They are identical, so the two expressions are proven equivalent.

6. *Problem:* Consider the following two truth tables. Are these tables equivalent?

a	b	result	b	a	result
T	T	T	F	T	F
T	F	F	F	F	T
F	T	T	T	F	T
F	F	T	T	T	T

Solution: At first glance, the two tables seem different: the first row in the left one is true, while the first row in the right one is false. However, carefully look at the values in the rows: these values are in different row orders. Further, the columns are also in a different order. We'll take the right truth table and convert it back to standard form step by step.

First, re-arrange the columns so that the variables are in incrementing order.

a	b	result
T	F	F
F	F	T
F	T	T
T	T	T

Next, rearrange the rows to match the row ordering of the standard table.

a	b	result
T	T	T
T	F	F
F	T	T
F	F	T

When ordered and arranged in the same manner, the truth tables are identical; thus, they are equivalent.



7. *Problem:* Convert the set expression $A' \cup B' \cap (A \cup B)'$ into a Boolean expression.

Solution: Replace each \cup with \vee , \cap with \wedge , and $'$ with \neg . Recall that in set notation a complement follows a variable or subexpression, while in Boolean notation the complement precedes the expression.

Thus, the Boolean equivalent expression is $\neg a \vee \neg b \wedge \neg(a \vee b)$.

8. *Problem:* Find a Boolean expression matching the following truth table:

a	b	c	result
T	T	T	F
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

Solution: Use the standard technique for disjunctive normal form. Start by crossing out all rows which have a false result.

a	b	c	result
T	T	T	F
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

Create a partial truth table by removing these crossed-out rows.

a	b	c	result
T	T	F	T
T	F	T	T
F	T	F	T
F	F	F	T

For each remaining row, create a disjunctive expression by ANDing together all the terms, with NOT in front of any indicated to be false.



a	b	c	Conjunction
T	T	F	$a \wedge b \wedge \neg c$
T	F	T	$a \wedge \neg b \wedge c$
F	T	F	$\neg a \wedge b \wedge \neg c$
F	F	F	$\neg a \wedge \neg b \wedge \neg c$

Finally, connect these subexpressions with OR operators to form the final expression: $(a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge \neg c)$.



A.6 Manipulating Logical Expressions

Exercises found in Chapter 6 on page 61.

1. *Problem:* Prove that DeMorgan's Law can be extended to additional terms. In other words, prove that $\neg(a \wedge b \wedge c) = \neg a \vee \neg b \vee \neg c$.

Solution: If a derivation which translates $\neg(a \wedge b \wedge c)$ into $\neg a \vee \neg b \vee \neg c$ can be devised, then it is likely this same derivation can apply to a larger number of terms.

1.	$\neg(a \wedge b \wedge c)$	Initial Expression
2.	$\neg((a \wedge b) \wedge c)$	DeMorgan's Law 8b
3.	$\neg(a \wedge b) \vee \neg c$	DeMorgan's Law 8b
4.	$\neg a \vee \neg b \vee \neg c$	Final Expression

If additional terms are needed, they can be "peeled off" incrementally by repeating step 2.

2. *Problem:* For each of the following Boolean expressions, indicate if it is a contradiction, a tautology, or just satisfiable.

- (a) $a \vee (b \wedge \neg a)$
- (b) $\neg(\neg a \vee \neg b) \wedge \neg(a \wedge b)$
- (c) $a \vee b \vee c \vee \neg(a \wedge b \wedge c)$

Solution: Contradiction, tautology, and satisfiable can be proven using truth tables.

- (a) $a \vee (b \wedge \neg a)$

We'll start with the framework for a two value (a and b) truth table.

a	b	$a \vee (b \wedge \neg a)$
T	T	
T	F	
F	T	
F	F	

First, solve the expression with $a = T$ and $b = T$. $T \vee \dots$ is always true, by the short-circuit rule, so the first row is true. The same reasoning holds for the second row. For the last two rows, we know that a is false, so we depend completely on $b \wedge \neg a$. Given that a is false, $\neg a$ will be true, so the AND operator reduces to just the value of b in these cases.



a	b	$a \vee (b \wedge \neg a)$
T	T	T
T	F	T
F	T	T
F	F	F

Some of the rows are true, and one is false. Therefore, this expression is satisfiable.

(b) $\neg(\neg a \vee \neg b) \wedge \neg(a \wedge b)$

We could create a truth table for this expression as-is, but a little simplification up front might make our job much easier.

1. $\neg(\neg a \vee \neg b) \wedge \neg(a \wedge b)$ Initial Expression
2. $\underline{\neg(\neg a \vee \neg b)} \wedge \neg(a \wedge b)$ DeMorgan's Law 8a
3. $\underline{\neg\neg a} \wedge \underline{\neg\neg b} \wedge \neg(a \wedge b)$ Double Negation Law 6
4. $(a \wedge b) \wedge \neg(a \wedge b)$ Complement Law 7b
5. F Final Expression

What does this mean for our truth table? It means that regardless of the value of a or b , the result will be false.

a	b	$\neg(\neg a \vee \neg b) \wedge \neg(a \wedge b)$
T	T	F
T	F	F
F	T	F
F	F	F

This expression is a contradiction.

(c) $a \vee b \vee c \vee \neg(a \wedge b \wedge c)$

Again, a little simplification up front can save a lot of time in the end.

1. $a \vee b \vee c \vee \neg(a \wedge b \wedge c)$ Initial Expression
2. $a \vee b \vee c \vee \underline{\neg(a \wedge b \wedge c)}$ DeMorgan's Law 8b
3. $a \vee b \vee c \vee \neg a \vee \neg b \vee \neg c$ Commutative Law 3a
4. $\underline{a \vee \neg a} \vee \underline{b \vee \neg b} \vee \underline{c \vee \neg c}$ Complement Law 7a
5. $T \vee T \vee T$ Identity Law 5c
6. T Final Expression

What does this mean for our truth table? It means that regardless of the value of a , b , or c , the result will be true.



a	b	c	$a \vee b \vee c \vee \neg(a \wedge b \wedge c)$
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	T
F	T	F	T
F	F	T	T
F	F	F	T

This expression is a tautology.

3. *Problem:* Simplify the Boolean expression $(a \vee b) \wedge (a \oplus \neg b)$.

Solution: Refer first to the definition of exclusive or (XOR) which tells us that $a \oplus b = (a \vee b) \wedge \neg(a \wedge b)$. We can rewrite this expression by inserting the definition of \oplus with $(a \vee b) \wedge ((a \vee \neg b) \wedge \neg(a \wedge \neg b))$.

1. $(a \vee b) \wedge ((a \vee \neg b) \wedge \neg(a \wedge \neg b))$ Initial Expression
2. $\underline{(a \vee b) \wedge (a \vee \neg b)} \wedge \neg(a \wedge \neg b)$ Distributive Law 4a
3. $a \vee \underline{(b \wedge \neg b)} \wedge \neg(a \wedge \neg b)$ Complement Law 7b
4. $a \vee \underline{F} \wedge \neg(a \wedge \neg b)$ Identity Law 5a
5. $a \wedge \underline{\neg(a \wedge \neg b)}$ DeMorgan's Law 8b
6. $a \wedge (\neg a \vee \underline{\neg \neg b})$ Double Negation Law 6
7. $a \wedge (\neg a \vee b)$ Distributive Law 4b
8. $\underline{(a \wedge \neg a)} \vee (a \wedge b)$ Complement Law 7b
9. $F \vee (a \wedge b)$ Identity Law 5a
10. $a \wedge b$ Final Expression

Note that in step 6 there may instead be a temptation to re-associate the parentheses to get rid of $a \wedge \neg a$. This would not be correct because parentheses cannot be re-associated between AND and OR. Those parentheses were created implicitly by step 5 which showed the transformation occurring as a single unit. In order to keep the order of operations correct, the parentheses had to be added.

Therefore, $(a \vee b) \wedge (a \oplus \neg b)$ can be simplified to $a \wedge b$.

4. *Problem:* Rewrite $(a \wedge b) \rightarrow (b \rightarrow c)$ without implications, then simplify.

Solution: Substitute the definition of implication: $a \rightarrow b = \neg a \vee b$. This gives us the expression $\neg(a \wedge b) \vee (b \rightarrow c)$. Substitute again to get the expression $\neg(a \wedge b) \vee (\neg b \vee c)$.



1.	$\neg(a \wedge b) \vee (\neg b \vee c)$	Initial Expression
2.	$\underline{\neg(a \wedge b)} \vee (\neg b \vee c)$	DeMorgan's Law 8b
3.	$(\neg a \vee \neg b) \vee (\neg b \vee c)$	Associative Laws
4.	$\neg a \vee \underline{\neg b} \vee \neg b \vee c$	Idempotent Law 1a
5.	$\neg a \vee \neg b \vee c$	Final Expression

You might notice the final expression can be translated into $a \rightarrow (b \rightarrow c)$.

5. *Problem:* Prove the implication law 10b (called contrapositive), that $x \rightarrow y = \neg y \rightarrow \neg x$.

Solution: We will apply various algebra laws, not with the intent of simplifying, but with the intent of reaching a specific goal. First, apply the definition of \rightarrow to both sides: $\neg x \vee y = \neg \neg y \vee \neg x$. We know that due to the commutative law we can reorder the left side to $y \vee \neg x$. Apply the double negation law to y , and there we are: $\neg \neg y \vee \neg x$.

6. *Problem:* Consider the implementation

`b = b == false`

- (a) Convert to logical form,
- (b) Simplify, and
- (c) Describe in words

Solution:

- (a) Convert to logical form

Note the difference between assignment and equality operators. As a logical expression, this can be represented as $b = b \leftrightarrow F$.

- (b) Simplify

Consider the right hand side for simplification. By the identity law 5e, $b \leftrightarrow F$ simplifies directly to $\neg b$. If, instead, a simplification using basic operators is desired, the following sequence can be followed:

1.	$b \leftrightarrow F$	Initial Expression, sub definition of equality
2.	$\neg(b \oplus F)$	Sub definition of XOR
3.	$\neg(\underline{b \oplus F})$	Identity Law 5g
4.	$\neg b$	Final Expression

Therefore $b = b \leftrightarrow F$ simplifies to $b = \neg b$. This can be implemented with

`b = !b`

- (c) Describe in words

This assignment statement has the action of inverting the value of `b` in the computer's memory.



7. *Problem:* Assume `response` and `value` are Boolean variables. For this implementation:

- (a) Create a truth table, and
- (b) Simplify

```
response = false
if (value == true) then
    response = true
else
    response = false
end if
```

Solution:

- (a) Create a truth table

We find that `response` is used only as an output, and `value` is used only as an input. Therefore, we have a brief truth table:

value	response
T	
F	

Filling in these two rows requires manual consideration of each possibility. If `value` is true, then `response` gets set to true, and the procedure ends. If, on the other hand, `value` is false, then `response` remains false. Therefore:

value	response
T	T
F	F

- (b) Simplify

The easiest simplification of this implementation is gleaned from viewing the truth table above. We can see that `response` takes on whatever is held in `value`, thus:

```
response = value
```

Is a simpler implementation.

8. *Problem:* Assume `self->active` and `STATUS` are Boolean variables. Note that `self->active` is one variable, the `->` does not carry any significance in terms of logical operators. For this implementation:

- (a) Create a truth table, and
- (b) Simplify

```

if (self->active != STATUS) then
    if ((self->active == false)
        && (STATUS == true)) then

        STATUS = false
    else if ((self->active == true)
        && (STATUS == false)) then

        STATUS = true
    end if
end if

```

Solution:

(a) Create a truth table

We find that `self->active` and `STATUS` are read as inputs, and that `STATUS` is also used as an output value. Therefore, we get a truth table like:

self->active	STATUS	STATUS
T	T	
T	F	
F	T	
F	F	

The entire implementation is surrounded by a condition that `self->active` and `STATUS` are different. If this is not so, then nothing happens, and `STATUS` retains its original value. So in the case where the two input values are equal, `STATUS` output is the same as input:

self->active	STATUS	STATUS
T	T	T
T	F	
F	T	
F	F	F

In the case where they aren't equal, each possibility is considered by the implementation: one must be true, and one must be false. Follow each of these conditions to fill in the remaining entries for `STATUS`:

self->active	STATUS	STATUS
T	T	T
T	F	T
F	T	F
F	F	F



(b) Simplify

Again, the best bet here is to look at the truth table. We find that the original input value of STATUS, although considered in the implementation, is actually irrelevant. Instead, in all cases, the output STATUS is the value of `self->active`. Therefore, we choose the new implementation:

```
STATUS = self->active
```



A.7 Decision Tables

Exercises found in Chapter 7 on page 71.

1. *Problem:* Consider selecting an appropriate college to attend. The student posits the following requirements:

- College must be in the Northwest region, unless it is a top ten college.
- College must offer computer science, or math, or both.
- Computer science program, if offered, must have at least one renowned faculty member.

Create a decision table to indicate if a college meets the requirements. Convert the decision table into Boolean expressions.

Solution: We must first define the various conditions and actions. Reading through the requirements list gives the following conditions: Northwest region, top ten college, offers CS, offers math, CS has at least one renowned faculty member.

There is only one action: apply to that college, or not.

Naively, with five Boolean conditions, this decision table would have $2^5 = 32$ condition columns. For space we make several abbreviations, in particular that a college having at least one renowned computer science faculty will be abbreviated as “rn. CS. fac.”

Cond. and Acts	Rules															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Northwest region	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
top ten college	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
offers math	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
offers CS	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
rn. CS. fac.	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Apply	Y	N	Y	Y	Y	N	N	N	Y	N	Y	Y	Y	N	N	N

Cond. and Acts	Rules															
	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
NW region	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
top ten college	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N
offers math	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N	N	N	N
offers CS	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
rn. CS. fac.	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Apply	Y	N	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N

There are several simplifications we can perform to reduce the size of the decision table: if a college does not have a computer science program, it cannot have any renowned computer



science faculty. Therefore, the condition offers CS is false and CS has at least one renowned faculty member is true is an impossible condition, and will be omitted. If a college is in the Northwest region, we don't care about top ten; so top ten is an indifferent condition in that case. If the college has a math department, then we can ignore whether or not it has a computer science department. However, if it does have a computer science department, we still must check to ensure it has renowned faculty. These are additional indifferent conditions. Finally, if the college is not in the northwest, and is not a top ten college, we can be indifferent about all remaining conditions.

Conditions and Actions	Rules							
	1	2	3	4	5	6	7	8
Northwest region	Y	Y	Y	N	N	N	N	-
top ten college	-	-	-	Y	Y	Y	N	-
offers math	Y	-	-	Y	-	-	-	N
offers CS	N	Y	Y	N	Y	Y	-	N
renowned CS faculty	-	Y	N	-	Y	N	-	-
Apply	Y	Y	N	Y	Y	N	N	N

By checking which outcomes have a true action, we can create a Boolean expression. Let n be Northwest region, t be top ten, m be offers math, c be offers CS, r be renowned CS faculty, and a be apply. Then $a = (n \wedge m \wedge \neg c) \vee (n \wedge c \wedge r) \vee (\neg n \wedge t \wedge m \wedge \neg c) \vee (\neg n \wedge t \wedge c \wedge r)$.

This expression can be simplified, if desired, to $a = (n \vee t) \wedge (m \vee c) \wedge (c \rightarrow r)$. Note that use of the implication operator to indicate that if a computer science program exists, it must have renowned faculty. If not, we don't care about renowned faculty.

2. *Problem:* Decompose the following multi-valued conditions into Boolean conditions. Convert the resulting decision table into a Boolean expression.

Conds. and Actions	Rules								
	1	2	3	4	5	6	7	8	9
Age	< 18	< 18	< 18	18 – 45	18 – 45	18 – 45	> 45	> 45	> 45
Membership	Non	Basic	Pr	Non	Basic	Pr	Non	Basic	Pr
Call from Trainer	N	N	N	N	Y	Y	Y	Y	Y

Solution: To create a Boolean decision table, each multi-valued condition (with more than two possibilities) should be translated into a condition for each possibility, with only one of the possibilities being true at a time. So we will have condition Age < 18, Age 18 – 45, Age > 45, Non-member, Basic member, and Premium member.



Conds. and Actions	Rules								
	1	2	3	4	5	6	7	8	9
Age < 18	Y	Y	Y	N	N	N	N	N	N
Age 18 – 45	N	N	N	Y	Y	Y	N	N	N
Age > 45	N	N	N	N	N	N	Y	Y	Y
Non-Member	Y	N	N	Y	N	N	Y	N	N
Basic Member	N	Y	N	N	Y	N	N	Y	N
Premium Member	N	N	Y	N	N	Y	N	N	Y
Call from Trainer	N	N	N	N	Y	Y	Y	Y	Y

Note that the number of condition columns does not change, thanks to the technique of eliminating impossible conditions.

To convert this table into a Boolean expression, we first identify variables for all conditions and actions. Let c be Age < 18, y be Age 18 – 45, and a be Age > 45. Let n be non-member, b be basic member, and p be premium member. The only action, t , is call from trainer. (We can't use c because it is already in use.)

By considering each column where the action is true, we can create the following expression: $t = (\neg c \wedge y \wedge \neg a \wedge \neg n \wedge b \wedge \neg p) \vee (\neg c \wedge y \wedge \neg a \wedge \neg n \wedge \neg b \wedge p) \vee (\neg c \wedge \neg y \wedge a \wedge n \wedge \neg b \wedge \neg p) \vee (\neg c \wedge \neg y \wedge a \wedge \neg n \wedge b \wedge \neg p) \vee (\neg c \wedge \neg y \wedge a \wedge \neg n \wedge \neg b \wedge p)$. Gag! However, we can apply our knowledge of impossible conditions to shorten this expression considerably. Note that this simplification is not generally true: it depends on certain permutations of conditions being impossible. In particular, we know that exactly one of the age conditions and exactly one of the membership conditions must be true. So if we specify just the true one, the other must be false. This reduces the expression to $t = (y \wedge b) \vee (y \wedge p) \vee (a \wedge n) \vee (a \wedge b) \vee (a \wedge p)$.

Additional reasoning can lead to additional simplification. Notice that when the age value a is true, all three membership types are accepted. We know that exactly one of these will be true, so that entire subexpression can be simplified to just a , giving us $t = (y \wedge b) \vee (y \wedge p) \vee a$.

In the first subexpressions, requiring that either a basic or premium membership is had is the same as requiring that a non-membership is NOT had, giving us $t = (y \wedge \neg n) \vee a$.

3. *Problem:* Create a decision table for the following statement:

A mailing is to be sent out to customers. The content of the mailing is about the current level of discounting and potential levels of discounting. The content is different for different types of customers. Customer Types A, B, and C get a normal letter except Customer Type C, who get a special letter. Any customer with 2 or more current lines or with a credit rating of 'X' gets a special paragraph added with an offer to subscribe to another level of discounting.

Convert the resulting decision table into a Boolean expressions.

Solution: First, identify conditions and actions. One condition is customer type. Types A, B, and C are indicated: however, there could be other types as well. We also need to know if the customer has 2 or more current lines, and if the credit rating is 'X'. Actions will be



to send a normal letter or special letter (or no letter), and to add a special offer paragraph. There is also the possibility, if there are customer types beyond A, B, or C, that we have a case where we would add a paragraph but don't have a letter to add it to. In that case, we'll ignore the paragraph.

We will use the multi-valued type customer with values A, B, C, and O. O will be for "other", any type not specified. For the action send letter, the values N for normal, S for special, or X for none will be used.

Conditions and Actions	Rules												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Customer Type	A	A	A	A	B	B	B	B	C	C	C	C	O
2 or more current lines?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	-
Credit rating 'X'?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	-
Send Letter	N	N	N	N	N	N	N	N	S	S	S	S	X
Special Offer	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	N

In order to convert this table into Boolean expressions, we must create a variable for each input and output. This involves decomposing multi-valued inputs and outputs. For customer type, we'll let the Boolean variables c_a , c_b , and c_c indicate the three known types of customers. We'll let t mean 2 or more current lines, and x mean credit rating of 'X'.

On the action side, the variables l_n and l_s will indicate which type of letter is sent (normal or special). We'll let s mean Special Offer.

There are then three expressions, two for the various send letter possibilities, and one for special offer.

Applying some intuitive simplification, we find the expressions are:

$$l_n = c_a \vee c_b$$

$$l_s = c_c$$

$$s = t \vee x$$

4. Problem: Create a decision table for the following statement:

If the package weight is less than 5 pounds, base shipping is \$4.00. If the package weight is 5 pounds to 10 pounds, base shipping is \$6.00. For packages more than 10 pounds, base shipping is \$10.00. If overnight shipping is selected, add \$20.00 to the shipping cost. If insurance is selected, double the base shipping price. Insurance is mandatory when overnight shipping is used. Packages 5 pounds or more should have a "heavy" label applied. If insurance is selected or the package is more than 10 pounds, have a "special freight" label applied.

Solution: The conditions are package weight (less than 5, 5 to 10, more than 10), overnight, and insurance. Actions are the shipping cost, whether a "heavy" label is applied, and whether a "special freight" label is applied.



For the package weight condition, we'll use S to mean less than 5, M to mean 5 to 10, and L to mean more than 10. Also note that insurance is mandatory with overnight shipping; this creates an impossible condition of insurance false and overnight true. Such a condition will be omitted from the table. We'll assume the "or" for the special freight label is an inclusive or.

Conditions and Actions	Rules								
	1	2	3	4	5	6	7	8	9
Package Weight	S	S	S	M	M	M	L	L	L
Overnight?	Y	N	N	Y	N	N	Y	N	N
Insurance?	Y	Y	N	Y	Y	N	Y	Y	N
Shipping cost	\$28	\$8	\$4	\$32	\$12	\$6	\$40	\$20	\$10
Heavy label	N	N	N	Y	Y	Y	Y	Y	Y
Special Freight label	Y	Y	N	Y	Y	N	Y	Y	Y

5. *Problem:* Consider the following decision table:

Conditions and Actions	Rules		
	1	2	3
Enrolled in Class	Y	Y	N
Passed Most Recent Exam	Y	-	N
Advertise Next Class	Y	N	Y
Send Checkup Email	Y	Y	N

- (a) Indicate all conditions which have contradictory actions.
- (b) Indicate all conditions which are undefined.
- (c) For all undefined conditions, is the condition probably impossible or simply undefined?

Solution: Look for missing or duplicated condition columns.

- (a) Indicate all conditions which have contradictory actions.
If a student is enrolled in class, and also passed their most recent exam, then two columns match (rule columns 1 and 2). This leads to a contradiction as to whether or not we should advertise the next class to them.
- (b) Indicate all conditions which are undefined.
If a student is not enrolled in the class but did pass their most recent exam, no column matches. In this case, the outcome is undefined.
- (c) For all undefined conditions, is the condition probably impossible or simply undefined?

We must decide if the most recent exam is in the class in question. If so, then it does not make sense for someone not enrolled in a class to have passed an exam in that class. Making that assumption, the condition would be impossible.



Consider the following decision table (called the original table) below:

Conditions and Actions	Rules							
	1	2	3	4	5	6	7	8
Under \$50	Y	Y	Y	Y	N	N	N	N
Pays by check	Y	Y	N	N	Y	Y	N	N
Pays by credit card	Y	N	Y	N	Y	N	Y	N
Call Supervisor	N	N	N	N	Y	Y	N	N
Check Photo ID	Y	Y	Y	N	Y	Y	Y	N
Proceed with sale	Y	Y	Y	Y	N	N	Y	Y

Assume that each purchase can be paid by only one method of payment: cash, check, or credit card.

6. *Problem:* Simplify the original table by applying indifferent conditions.

Solution: To find indifferent conditions, search for similar actions. For example, rules 1, 2, 3, and 7 have the same set of outcomes. Rules 4 and 8 have the same set of outcomes. Rules 5 and 6 have the same set of outcomes. For each set, look at the conditions to see if certain combinations are covered.

Rules 1 and 2 cover all possible values of pay by credit card, so these can be condensed into one rule. Rules 4 and 8, and also 3 and 7, cover all possible values of under \$50, so these can be condensed into one rule for each pair. Rules 5 and 6 cover all possible values of pays by credit card, so these can be condensed into one rule. Note that rules 1, 2, 5, and 6 cannot be all combined because other aspects of the conditions (Under \$50?) and actions (Call supervisor and proceed with sale) are different.

This allows us to condense the table to just four condition rule columns:

Conditions and Actions	Rules			
	1	2	3	4
Under \$50	Y	-	-	N
Pays by check	Y	N	N	Y
Pays by credit card	-	Y	N	-
Call Supervisor	N	N	N	Y
Check Photo ID	Y	Y	N	Y
Proceed with sale	Y	Y	Y	N

7. *Problem:* Simplify the original table by detecting impossible conditions.

Solution: The presence of impossible conditions is indicated by the phrase that tells us that only ONE method of payment will be used. If both check and credit card are false, we can

assume cash is used. However, it is not possible that both check and credit card can be true, as this would indicate two methods of payment. Any columns with both these values true can be removed as impossible.

Looking at the original table, rules 1 and 5 are thus impossible conditions and can be removed. This shortens the table to six rules.

Conditions and Actions	Rules					
	1	2	3	4	5	6
Under \$50	Y	Y	Y	N	N	N
Pays by check	Y	N	N	Y	N	N
Pays by credit card	N	Y	N	N	Y	N
Call Supervisor	N	N	N	Y	N	N
Check Photo ID	Y	Y	N	Y	Y	N
Proceed with sale	Y	Y	Y	N	Y	Y

8. *Problem:* Simplify the original table by applying multi-valued conditions.

Solution: One clue that a multi-valued condition might be possible if the statement that only ONE method of payment will be used. Thus, after eliminating impossible conditions (as in the previous problem), we can reduce the payment condition to a single row, with three values: C for cash, H for check, and R for credit card.

Conditions and Actions	Rules					
	1	2	3	4	5	6
Under \$50	Y	Y	Y	N	N	N
Payment Method	H	R	C	H	R	C
Call Supervisor	N	N	N	Y	N	N
Check Photo ID	Y	Y	N	Y	Y	N
Proceed with sale	Y	Y	Y	N	Y	Y

Finally, it is certainly possible to apply all three techniques to create a decision table which is as simple and compact as possible:

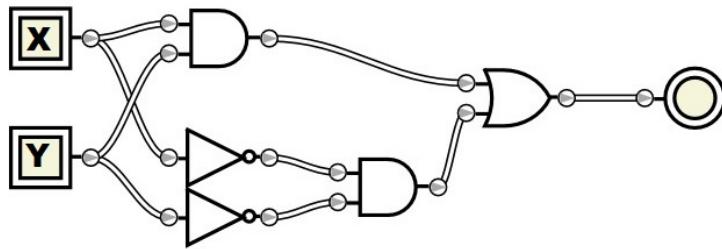
Conditions and Actions	Rules				
	1	2	3	4	5
Under \$50	Y	Y	-	N	N
Payment Method	H	R	C	H	R
Call Supervisor	N	N	N	Y	N
Check Photo ID	Y	Y	N	Y	Y
Proceed with sale	Y	Y	Y	N	Y



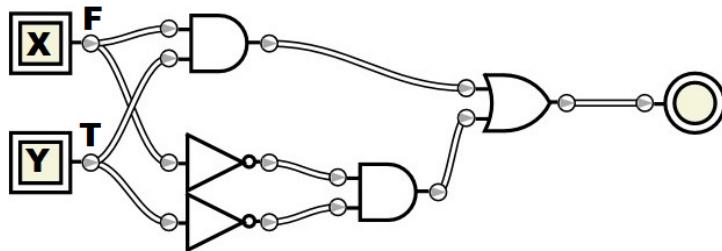
A.8 Logic Circuits

Exercises found in Chapter 8 on page 87.

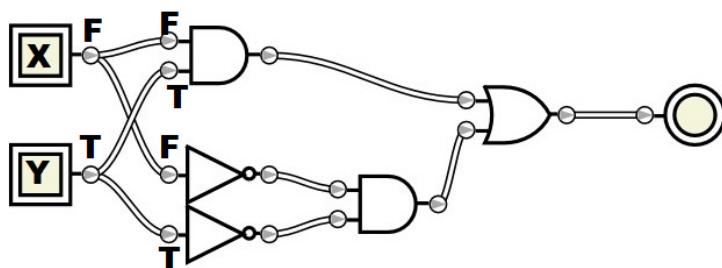
1. *Problem:* Determine the output of the given circuit if x is false and y is true.



Solution: Place the input values with the associated input gates.

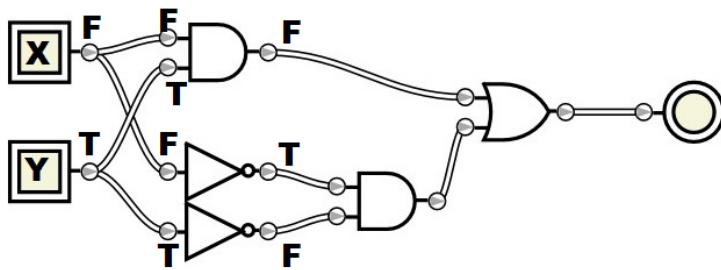


Trace along the wires to propagate values.

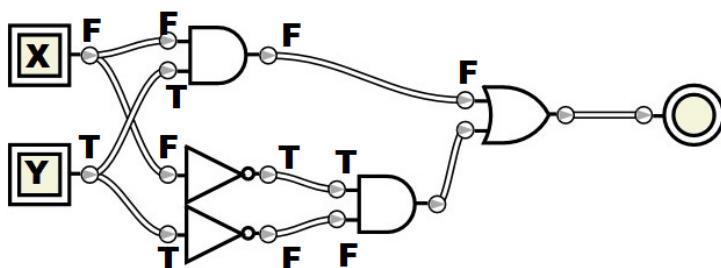


All gates which have all their inputs available can be solved, producing an output. The leftmost AND and the two NOT gates meet this criteria. For the AND gate, false and true will be false. The NOT gates invert their respective inputs.

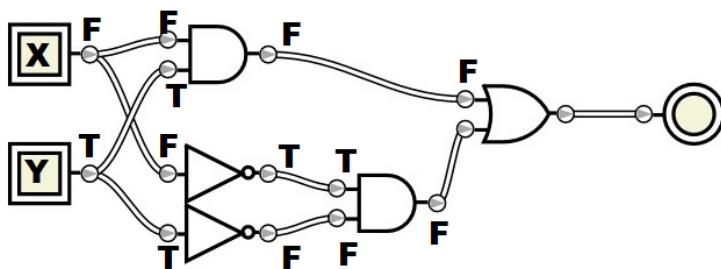




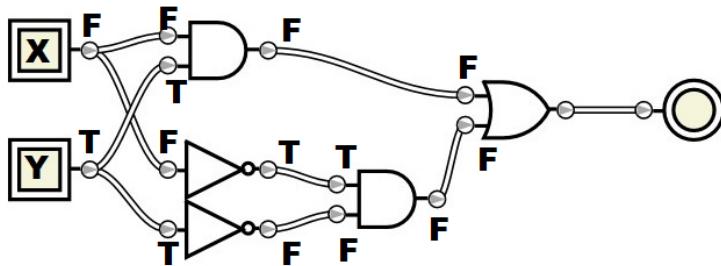
Continue tracing.



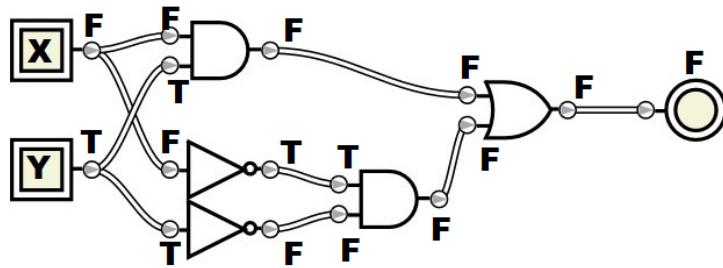
The topmost OR gate has one value ready, but not the other. So we cannot solve the OR gate yet. The AND gate on the bottom, however, has both inputs ready, so it can be solved. True and false will again be false.



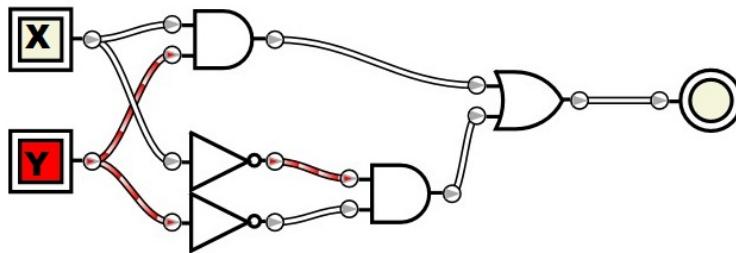
Propagate that result.



The final OR gate can now be solved. False or false yields false, so the result of the entire circuit is false for the given inputs.



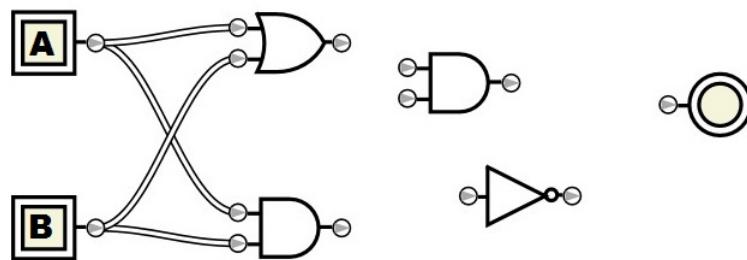
Alternative Solution: Enter the circuit into a simulation program, and select the appropriate input values.



This shows the inputs configured as specified, and the output of the circuit is false.

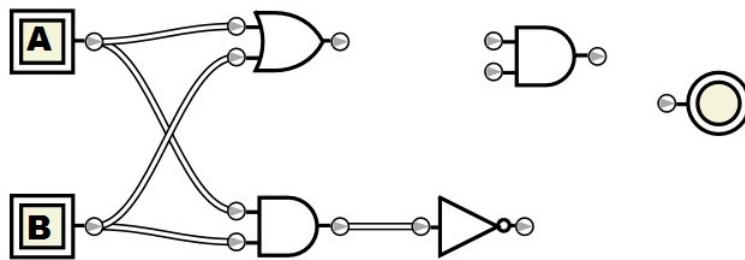
2. *Problem:* Show the definition of exclusive or, \oplus , in a circuit using only basic gates.

Solution: Recall that exclusive or (XOR) is defined as “one or the other, but not both.” A previous chapter gave the definition $a \oplus b = (a \vee b) \wedge \neg(a \wedge b)$. We can create a circuit with one OR, two ANDs, and one NOT. From inside-out, the innermost operations are $a \vee b$ and $a \wedge b$, so we wire these from the inputs.

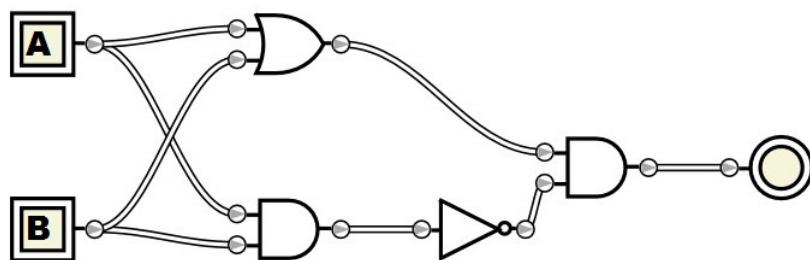


Next, the output of the AND part (on the bottom), is connected to a NOT gate.



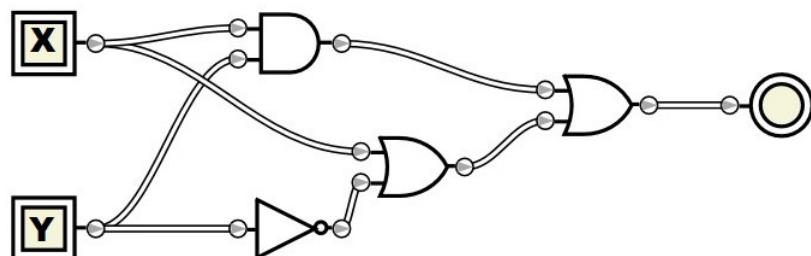


Finally, both results are connected with the final AND gate. The top part reflects “one or the other” and the bottom part reflects “not both”; these are combined with an AND to indicate that both conditions must be met. Here is the final circuit that represents exclusive OR using only basic gates.

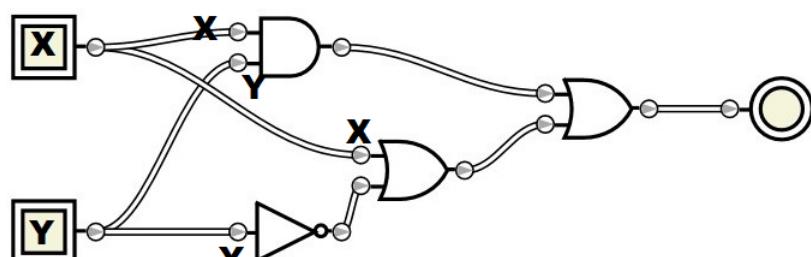


If compound gates were allowed, the AND-NOT sequence on the bottom could be replaced with a NAND gate.

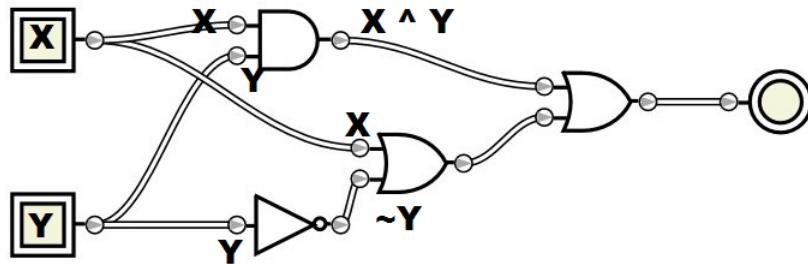
3. *Problem:* Create a Boolean expression for the circuit shown.



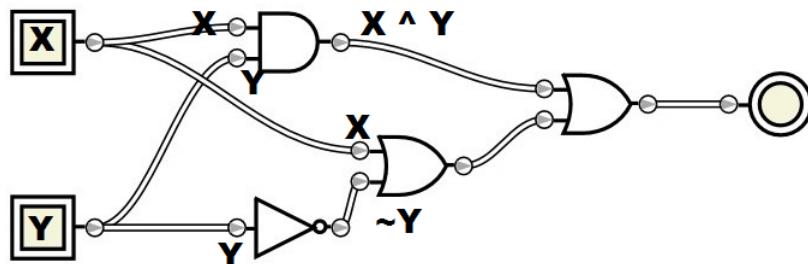
Solution: Start by tracing the values from the origins.



The AND gate on top, and the NOT gate on bottom have all their inputs available first. The OR gate has one input available, but not the other, so we can't complete it yet.

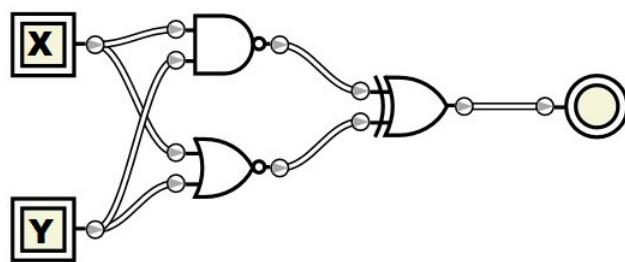


Next, we can complete the bottom OR gate, which makes the input subexpressions available for both inputs of the final OR gate.



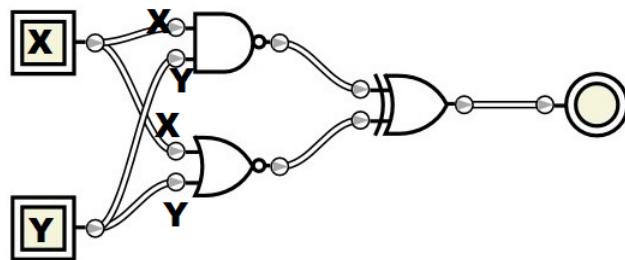
Combining these with an OR (be sure to use parentheses to enforce order of operations) gives us the expression $(x \wedge y) \vee (x \vee \neg y)$.

4. *Problem:* Create a Boolean expression for the circuit shown.



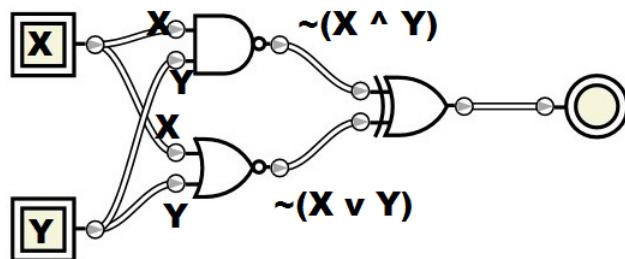
Solution: Start by tracing the values from the origins.





Two gates can be solved. Be careful to identify them correctly! The top gate is the NAND gate (this can be confirmed by the small circle following it). The bottom gate is the NOR gate. Likewise, look for the small circle after the gate symbol. The NAND gate is equivalent to AND followed by NOT; likewise, the NOR gate is equivalent to OR followed by NOT.

There are no symbols in expressions for NAND or NOR, so they must be expanded using their definitions.



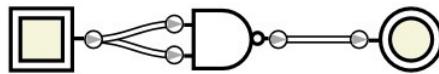
The final gate is also a compound gate, the XOR (exclusive OR) gate. We could expand the definition of exclusive OR, however, there is an expression symbol for XOR: \oplus . Using this symbol directly in the final expression is both appropriate and helpful, since it results in a smaller expression.

The final expression is $\neg(x \wedge y) \oplus \neg(x \vee y)$.

5. *Problem:* Prove that NAND is universal by using NAND gates to construct circuits which simulate:

- (a) an AND gate
- (b) an OR gate
- (c) a NOT gate

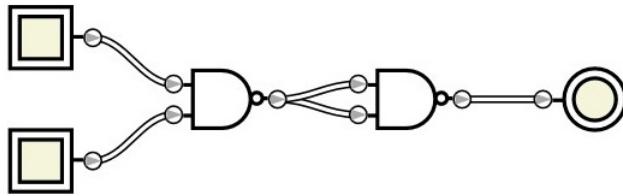
Solution: The easiest way to begin with this problem is to attack the NOT gate. We know that NAND is defined as AND followed by NOT. An AND gate which receives both inputs from the same source becomes simply an identity gate: if both inputs are true, the output is true. If both inputs are false, the output is false. Thus, if we attach the same input to both inputs of a NAND gate, the AND gate falls away and it behaves like a NOT gate.



We can prove this behavior using Boolean algebra laws.

1. $\neg(a \wedge a)$ Initial Expression
2. $\neg(\underline{a \wedge a})$ Idempotent Law 1b
3. $\neg a$ Final Expression

With this NOT simulator in place, we can easily tackle the creation of an AND gate. A NAND gate is AND followed by NOT. Adding a second NOT after the NAND gate will cause the two NOTs to cancel, leaving just an AND gate. Therefore, if we attach our definition of NOT above to a NAND gate, we should get an AND equivalent.



We can prove this behavior using Boolean algebra laws.

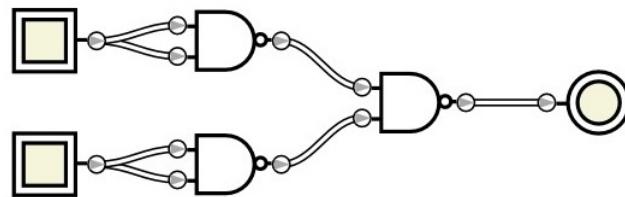
1. $\neg(\neg(a \wedge b) \wedge \neg(a \wedge b))$ Initial Expression
2. $\neg(\underline{\neg(a \wedge b)} \wedge \underline{\neg(a \wedge b)})$ Idempotent Law 1b
3. $\neg\neg(a \wedge b)$ Double Negation Law 6
4. $a \wedge b$ Final Expression

We have now reasoned our way into AND and NOT. What about OR? To create an OR gate, start with some Boolean algebra laws to create an equivalent expression using only AND and NOT. In a sense, we work this one backwards to find an expression constructed of gates we are allowed to use.

1. $a \vee b$ Initial Expression
2. $a \vee b$ Double Negation Law 6 (applied to both sides)
3. $\neg\neg a \vee \neg\neg b$ DeMorgan's Law 8b
4. $\neg(\neg a \wedge \neg b)$ Final Expression

We can convert this expression into a circuit. Using the NOT/NAND equivalence shown, first implement $\neg a$ and $\neg b$. Then connect them with a NAND gate to take care of the AND and the final NOT.

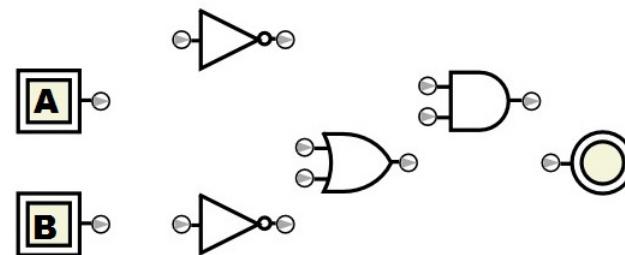




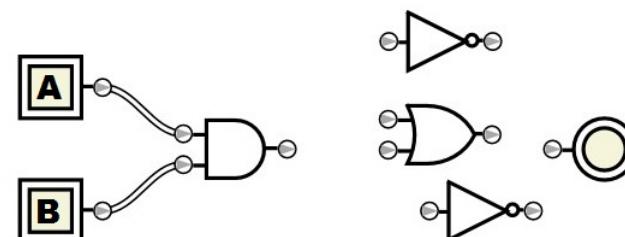
With the ability to simulate AND, OR, and NOT, all possible circuits can be now be simulated.

6. *Problem:* Create a logic circuit for the expression $\neg(\neg a \vee (a \wedge b))$.

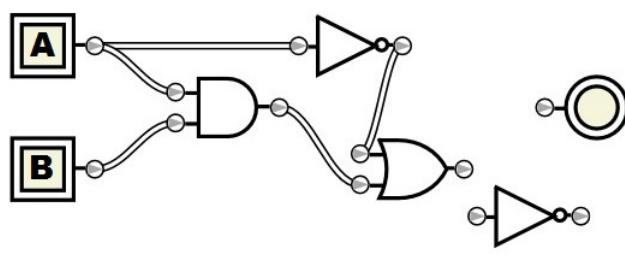
Solution: Start by laying out all the gates that will be needed: two NOTs, an OR, and an AND.



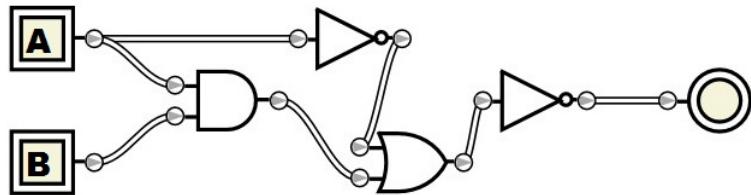
Following order of operations, proceed from the inside of the expression out. The innermost parentheses group indicates the subexpression $a \wedge b$, so connect that first.



Moving outwards, a $\neg a$ is connected to the previous subexpression using an OR.



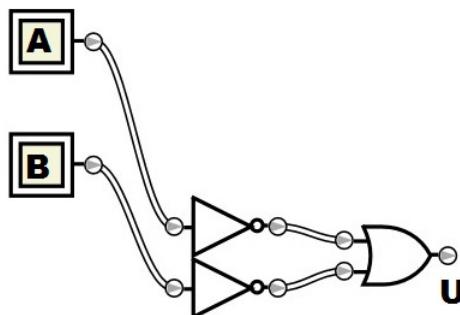
Finally, the entire result so far is passed through an outer NOT gate, which produces the final result:



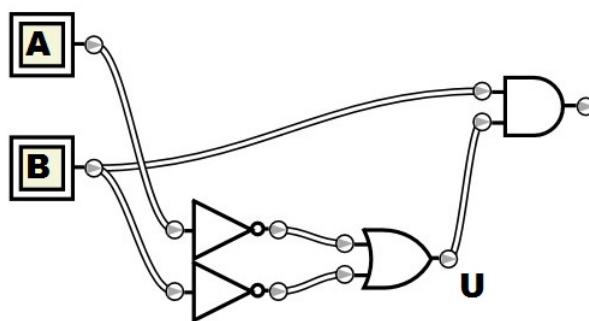
7. Problem: Create a logic circuit for the expression $\neg(\neg a \vee \neg b) \vee (b \wedge (\neg a \vee \neg b))$. Take advantage of reuse to avoid unnecessary gates.

Solution: First, identify repeated subexpressions which can be reused. The subexpression $(\neg a \vee \neg b)$ appears twice. Let $u = \neg a \vee \neg b$, and update the original expression to take advantage of this definition. The modified expression is $\neg u \vee (b \wedge u)$.

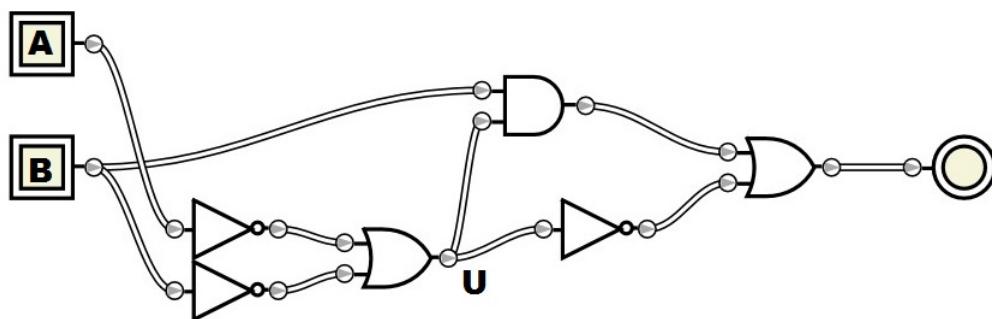
First, create a circuit which represents the subexpression defined as u . This circuit inverts a and b and combines them with an OR.



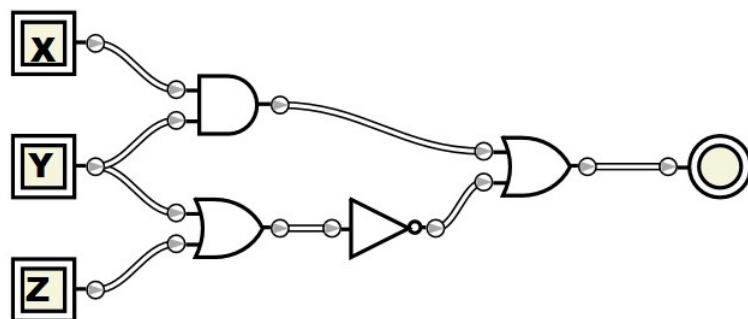
As usual, work from the inside-out, taking advantage of u wherever it is appropriate. First the $b \wedge u$ subexpression is created.



This result can be combined with $\neg u$ using an OR to create the final circuit:



8. Problem: Convert this circuit into disjunctive normal form.

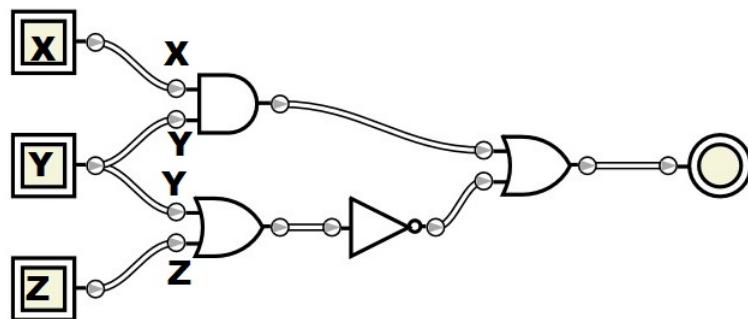


Solution: A four step process will be employed.

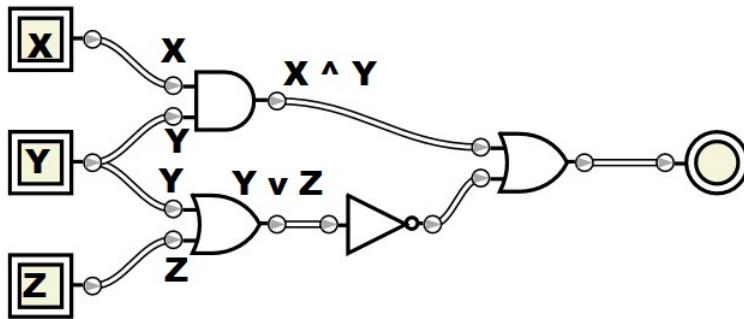
- Convert the circuit to a Boolean expression.
- Create the truth table for the Boolean expression.
- Create a disjunctive normal form Boolean expression from the truth table.
- Convert this expression into a circuit.

It is also possible to find the truth table directly from the circuit by simulating all possible inputs, thus combining the first two steps.

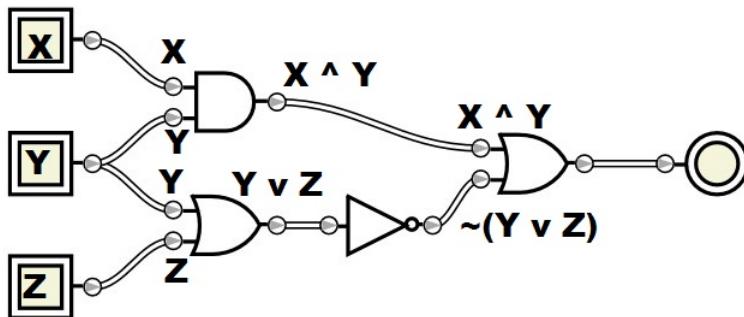
First, convert the circuit to a Boolean expression. Trace through the circuit along the wires.



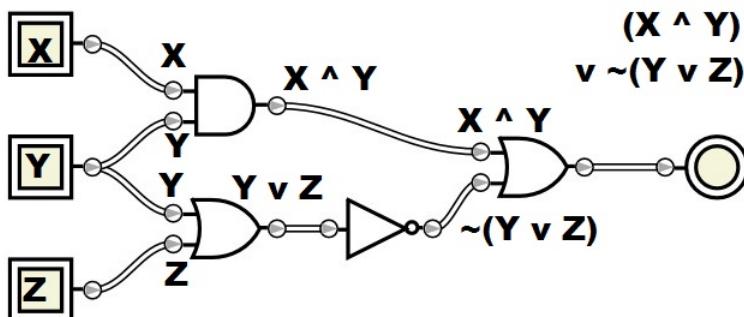
The AND (top) and OR (bottom) occur first, constructing subexpressions $x \wedge y$ and $y \vee z$ respectively.



The NOT (bottom) applies to the OR'd subexpression, and we prepare for the final operator.



Applying the final OR, we determine the equivalent Boolean expression.



The expression for this circuit is $(x \wedge y) \vee \neg(y \vee z)$.

Next, create a truth table for the expression.



x	y	z	$(x \wedge y) \vee \neg(y \vee z)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	T

Following the procedure for converting a truth table back into a Boolean expression in disjunctive normal form, cross out all the false rows.

x	y	z	$(x \wedge y) \vee \neg(y \vee z)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	T

For each remaining row, generate an expression by ANDing each condition.

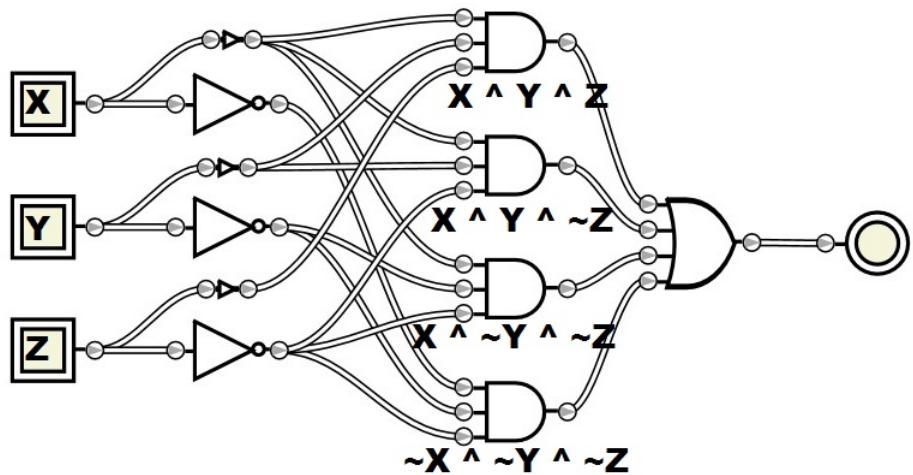
x	y	z	Conjunction
T	T	T	$x \wedge y \wedge z$
T	T	F	$x \wedge y \wedge \neg z$
T	F	F	$x \wedge \neg y \wedge \neg z$
F	F	F	$\neg x \wedge \neg y \wedge \neg z$

Combine all the conjunctions with ORs to create an expression in disjunctive normal form:
 $(x \wedge y \wedge z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge \neg z)$

To implement this expression in a circuit, place the NOTs first, followed by the ANDs, followed by the master OR. It is acceptable to reuse the NOT results. For example $\neg z$ is used in several subexpressions. Note that in this circuit, the small triangles (called buffer gates) do not change the value of their respective inputs, they are simply used to organize the wiring.

The final circuit mirrors the disjunctive normal form expression:





A.9 Number Systems

Exercises found in Chapter 9 on page 99.

1. *Problem:* An IPv6 address is comprised of eight blocks of four hexadecimal digits. How many bits long is an IPv6 address?

Solution: First, determine how many hexadecimal digits are in an IPv6 address: $8 * 4 = 32$. Each hexadecimal digit is equivalent to four binary digits (bits). We can confirm this by noting a hexadecimal digit has 16 possibilities, and, by the multiplicity counting rule, four bits have $2^4 = 16$ possibilities as well. Finally $32 * 4 = 128$. Therefore, an IPv6 address is 128 bits long.

2. *Problem:* Convert the octal number 737_8 into hexadecimal.

Solution: Both octal and hexadecimal are representations of binary. Thus, the easiest approach is to do the conversion via binary. Recall that each octal digit represents three bits. Using either counting, or the octal-binary conversion table, we find: $737_8 = 111\ 011\ 111_2$.

Rearranging the bits to allow for four-bit blocks gives us $1\ 1101\ 1111_2$. The number of bits is not evenly divisible by four (causing the odd one out). Like commas separating blocks of three in decimal numbers, we split these numbers into blocks from the right, and if necessary, pad out the number with 0s on the left. So $1\ 1101\ 1111_2 = 0001\ 1101\ 1111_2$.

Finally, substitute the appropriate hexadecimal digit for each four-bit block, using either counting or the hexadecimal-binary conversion table. This gives us $1DF_{16}$.

Alternative Solution: Convert via decimal. The octal to decimal conversion is accomplished by multiplying each digit in the octal number by its place value.

$$\begin{array}{r} 7 & 3 & 7 \\ 8^2 & 8^1 & 8^0 \end{array}$$

Then:

$$\begin{array}{rcl} 7 * 8^2 & = & 448 \\ 3 * 8^1 & = & 24 \\ 7 * 8^0 & = & 7 \\ \hline & & 479 \end{array}$$

The total is $448 + 24 + 7 = 479_{10}$. Next, convert from decimal to hexadecimal.

Divide by 16 (the hexadecimal base), recalling the remainders form the new number.

$$\begin{array}{rcl} 479 \div 16 & = & 29r15 \\ 29 \div 16 & = & 1r13 \\ 1 \div 16 & = & 0r01 \end{array}$$



This gives us the sequence 15, 13, 1. The first remainder is the rightmost digit, so the hexadecimal number is $1DF_{16}$.

3. *Problem:* Convert the binary number 101110_2 into hexadecimal.

Solution: A hexadecimal digit is equivalent to four bits. Break the binary number into four bit blocks, starting from the right. This gives us $10\ 1110_2$. Pad zeros on the left to create four bit blocks, giving us $0010\ 1110_2$. Substitute the appropriate hexadecimal digit for each four-bit block: $2E_{16}$.

4. *Problem:* Convert the hexadecimal number $5D6B_{16}$ into decimal.

Solution: Set up a conversion table with each digit and the place value, as powers of the base.

$$\begin{array}{cccc} 5 & D & 6 & B \\ 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$

Substitute the appropriate values for letters.

$$\begin{array}{cccc} 5 & \boxed{13} & 6 & \boxed{11} \\ 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$

$$\begin{array}{rcl} 5 * 16^3 & = & 20480 \\ 13 * 16^2 & = & 3328 \\ 6 * 16^1 & = & 96 \\ 11 * 16^0 & = & 11 \\ \hline & & 23915 \end{array}$$

The sum of the multiplications gives the conversion result: $5D6B_{16} = 23915_{10}$.

Alternative Solution: Starting with the leftmost digit, accumulate the sum by adding the current digit, and if more digits remain, multiply the current sum by the base (16, in this case).

- 5 (leftmost digit)
- $5 * 16 = 80$ (more digits remain)
- $80 + 13 = 93$ ($D = 13$)
- $93 * 16 = 1488$ (more digits remain)
- $1488 + 6 = 1494$
- $1494 * 16 = 23904$ (more digits remain)
- $23904 + 11 = 23915$ (that's the end, no more digits)



Therefore, $5D6B_{16} = 23915_{10}$

5. *Problem:* Convert the number 4033_5 into decimal.

Solution: Base 5 is not a commonly used base. However, all the usual conversion techniques apply. We can start by setting up a conversion table.

$$\begin{array}{cccc} 4 & 0 & 3 & 3 \\ 5^3 & 5^2 & 5^1 & 5^0 \end{array}$$

Determine the powers of five, multiply each digit by its place value and add the result.

$$\begin{array}{rcl} 4 * 5^3 & = & 500 \\ 0 * 5^2 & = & 0 \\ 3 * 5^1 & = & 15 \\ 3 * 5^0 & = & 3 \\ \hline & & 518 \end{array}$$

Thus, $4033_5 = 518_{10}$.

Alternative Solution: Starting with the leftmost digit, accumulate the sum by adding the current digit, and if more digits remain, multiply the current sum by the base (5, in this case).

- 4 (leftmost digit)
- $4 * 5 = 20$ (more digits remain)
- $20 + 0 = 20$
- $20 * 5 = 100$ (more digits remain)
- $100 + 3 = 103$
- $103 * 5 = 515$ (more digits remain)
- $515 + 3 = 518$ (that's the end, no more digits)

Thus, $4033_5 = 518_{10}$.

Check: In most of these problems, a base-converting calculator can be employed to check our answer. However, many base-converting calculators only operate on common bases, such as 2, 8, 10, and 16. In order to check the answer, we may convert back from decimal into base 5.

The decimal result is 518_{10} . Divide by the target base (5) and record the remainders.

$$\begin{array}{rcl} 518 \div 5 & = & 103r3 \\ 103 \div 5 & = & 20r3 \\ 20 \div 5 & = & 4r0 \\ 4 \div 5 & = & 0r4 \quad (\text{stop because quotient is now 0}) \end{array}$$



In this process, the first remainder produced is the rightmost digit. Thus, the conversion yields $518_{10} = 4033_5$. This matches the original problem, so we are confident the result is correct.

6. *Problem:* Convert the number 51.15_8 into decimal.

Solution: Notice the presence of the decimal place. This tells us where, in our conversion chart, the power will switch to negative.

$$\begin{array}{cccc} 5 & 1 & 1 & 5 \\ 8^1 & 8^0 & 8^{-1} & 8^{-2} \end{array}$$

Once the powers have been assigned, the decimal place can be forgotten. If needed, the definition of negative powers can be applied.

$$\begin{array}{cccc} 5 & 1 & 1 & 5 \\ 8^1 & 8^0 & \frac{1}{8^1} & \frac{1}{8^2} \end{array}$$

Calculate the values of the places. Note that the number of decimal places may be substantial. In the case of unusual bases, such as base 3, some of the decimals repeat endlessly. In that case, rounding to an appropriate number of places is usually acceptable, but the answer is then an approximation and not exact. If an exact representation is desired, fractions can be used.

$$\begin{array}{rcl} 5 * 8^1 & = & 40.000000 \\ 1 * 8^0 & = & 1.000000 \\ 1 * 8^{-1} & = & 0.125000 \\ 5 * 8^{-2} & = & 0.078125 \\ \hline & & 41.203125 \end{array}$$

Summing the results indicates the conversion $51.15_8 = 41.203125_{10}$.

Using Fractions: Continue from the powers chart:

$$\begin{array}{cccc} 5 & 1 & 1 & 5 \\ 8^1 & 8^0 & \frac{1}{8^1} & \frac{1}{8^2} \end{array}$$

Compute the whole number portion in the usual way, but leave the decimal portion as fractions instead.

$$\begin{array}{rcl} 5 * 8^1 & = & 40 \\ 1 * 8^0 & = & 1 \\ 1 * 8^{-1} & = & \frac{1}{8} \\ 5 * 8^{-2} & = & \frac{5}{64} \end{array}$$



In order to add fractions, a common denominator must be found. The easiest common denominator in this case is 64. This gives the sum $40 + 1 + \frac{8}{64} + \frac{5}{64} = 41\frac{13}{64}$.

7. *Problem:* Convert the decimal number 53.3_{10} into binary.

Solution: Convert the whole number and fractional portion separately. The whole number portion can be converted by repeated divisions and retaining the remainders.

$$\begin{aligned}
 53 \div 2 &= 26r1 \\
 26 \div 2 &= 13r0 \\
 13 \div 2 &= 6r1 \\
 6 \div 2 &= 3r0 \\
 3 \div 2 &= 1r1 \\
 1 \div 2 &= 0r1 \quad (\text{quotient is zero, so stop})
 \end{aligned}$$

The first remainder found is the rightmost digit, so the whole number portion $53_{10} = 110101_2$.

On to the fractional part. Begin with 0.3_{10} . Multiply by the target base (2) and extract the whole number part. Continue until no fractional part remains.

$$\begin{aligned}
 0.3 * 2 &= 0.6 \text{ (digit 0)} \\
 0.6 * 2 &= 1.2 \text{ (digit 1)} \\
 0.2 * 2 &= 0.4 \text{ (digit 0)} \\
 0.4 * 2 &= 0.8 \text{ (digit 0)} \\
 0.8 * 2 &= 1.6 \text{ (digit 1)}
 \end{aligned}$$

At this point, we would be back to $0.6 * 2 = \dots$, indicating the sequence will repeat forever. Thus, $53.3_{10} = 110101.0\overline{1001}_2 = 110101.010011001100110011001\dots_2$. In base 2, this value is repeating and thus cannot be exactly represented with any number of digits. Most commonly, a finite number of bits are available and so the number is rounded slightly. This rounding is often not noticed but can occasionally lead to certain computational errors.

8. *Problem:* Order the numbers from least to greatest: 123_8 , 123_4 , 123_{16} , 123_{10} .

Solution: Convert all values into the same base, and then compare in the usual way. Two of the values are in bases easily converted to binary (base 8 and 16). What about base 4? Four is a power of two, so base 4 is likewise a straightforward conversion to and from binary:

Base 4	Binary
0	00
1	01
2	10
3	11



The easiest conversion is probably to convert everything into binary and compare from there, as three of the four values have immediate binary conversions via substitution. Specifically:

- $123_8 = 001\ 010\ 011_2 = 1010011_2$
- $123_4 = 01\ 10\ 11_2 = 11011_2$
- $123_{16} = 0001\ 0010\ 0011_2 = 100100011_2$

The final conversion is 123_{10} into binary. Divide by the target base (2) and retain remainders.

$$\begin{array}{rcl}
 123 \div 2 & = & 61r1 \\
 61 \div 2 & = & 30r1 \\
 30 \div 2 & = & 15r0 \\
 15 \div 2 & = & 7r1 \\
 7 \div 2 & = & 3r1 \\
 3 \div 2 & = & 1r1 \\
 1 \div 2 & = & 0r1 \quad (\text{stop at quotient 0})
 \end{array}$$

The remainders produce the rightmost digit first, so $123_{10} = 1111011_2$.

In the absence of any leading zeros, whole numbers with more digits are larger.

- $123_8 = 1010011_2$ (7 digits)
- $123_4 = 11011_2$ (5 digits)
- $123_{16} = 100100011_2$ (9 digits)
- $123_{10} = 1111011_2$ (7 digits)

Numbers with the same digit count can be compared in the usual way, left to right, with the first non-tied digit being used to determine the larger number. In order from least to greatest:

- $123_4 = 11011_2$ (5 digits)
- $123_8 = 1010011_2$ (7 digits)
- $123_{10} = 1111011_2$ (7 digits)
- $123_{16} = 100100011_2$ (9 digits)

Alternative Solution: Consider the fact that for any two numbers with the same digits and different bases, the numbers can be ordered by their bases. That is, if $b < c$ then $a_b < a_c$. Given this rule, and the fact that all the numbers have the same digits, they can be simply ordered from least to greatest by their bases: $123_4, 123_8, 123_{10}, 123_{16}$.



A.10 Integer Numbers

Exercises found in Chapter 10 on page 111.

1. *Problem:* Show how the computer performs the unsigned addition $45_{10} + 17_{10}$ in eight bit binary.

Solution: First, convert each value into binary. $45_{10} = 101101_2$ and $17_{10} = 10001_2$. In this case, a fixed width of eight bits has been specified. Pad out each number on the left with 0s until eight bits is reached. If a number is more than eight bits, it cannot be represented. This gives us $0010\ 1101_2$ and $0001\ 0001_2$.

$$\begin{array}{r}
 & & & & & & ^1 & \\
 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
 \hline
 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0
 \end{array}$$

The result, $0011\ 1110_2$ can be translated back into decimal, to confirm that $45_{10} + 17_{10} = 62_{10}$.

2. *Problem:* What happens when the computer attempts to perform the unsigned addition $201_{10} + 99_{10}$ in unsigned eight bit binary? How can the error be detected?

Solution: First, convert each value into binary and pad to eight bits, if needed: $201_{10} = 1100\ 1001_2$ and $99_{10} = 0110\ 0011_2$. Note that the value 201_{10} , when translated into eight bit binary, has a left-most bit of 1. Is this a problem? Does it indicate a negative? No, it is fine: the problem specified unsigned eight bit binary. If Two's Complement had been specified, then we would stop and say that 201_{10} cannot be represented in eight bit Two's Complement.

$$\begin{array}{r}
 & & & & & & ^1 & ^1 \\
 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
 \hline
 10 & 0 & 1 & 0 & 1 & 1 & 0 & 0
 \end{array}$$

Only eight bits are available for storage, so the extra bit (which triggers an overflow condition) does not appear in the eight bit result: $1100\ 1001_2 + 0110\ 0011_2 = 0010\ 1100_2$ with overflow. The error in this result can be detected by noting the presence of the overflow condition.

3. *Problem:* Show how the computer represents -77_{10} in eight bit Two's Complement.

Solution: Note that the value is negative. First, represent the value in binary, padding to the appropriate number of bits as needed: $-77_{10} = -0100\ 1101_2$. If the original value



were positive, this would be the final Two's Complement result. However, to represent a negative number in Two's Complement, we invert and add one.

Inverting 0100 1101 gives 1011 0010.

$$\begin{array}{ccccccccc}
 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
 \hline
 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1
 \end{array}$$

Thus, $-77_{10} = 1011\ 0011_{2C}$.

Check: Convert $1011\ 0011_2$ into decimal. Note the leftmost bit is 1, therefore, the final number will be negative. In order to find the absolute value, invert, and add one.

Inverting 1011 0011 gives 0100 1100.

$$\begin{array}{ccccccccc}
 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
 \hline
 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1
 \end{array}$$

Finally, $-0100\ 1101_2 = -77_{10}$. The result is confirmed.

Alternative Check: Convert $1011\ 0011_2$ into decimal. Use the Two's Complement to decimal shorthand. Perform a binary to decimal conversion in the usual way, except that the value of the leftmost bit is negative.

$$1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \\ -2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

Add each place.

$$\begin{array}{rcl}
 1 * -2^7 & = & -128 \\
 0 * 2^6 & = & 0 \\
 1 * 2^5 & = & 32 \\
 1 * 2^4 & = & 16 \\
 0 * 2^3 & = & 0 \\
 0 * 2^2 & = & 0 \\
 1 * 2^1 & = & 2 \\
 1 * 2^0 & = & 1 \\
 \hline
 & & -77
 \end{array}$$

Sum the results, yielding -77_{10} . The result is confirmed.



4. *Problem:* Show how the computer performs the subtraction $13_{10} - 23_{10}$ in eight bit Two's Complement.

Solution: Recall that the computer does not actually perform subtraction. Instead, the right-hand value is negated (negative sign put in front), and the operation is addition. This gives a modified problem of $13_{10} + (-23_{10})$.

Convert each value to binary, padding bits as needed. $0000\ 1101_2 + (-0001\ 0111_2)$. Convert both values into Two's Complement. The leftmost value is positive, so nothing changes. The rightmost value is negative, so invert and add one. The problem is now $0000\ 1101_{2C} + 1110\ 1001_{2C}$.

$$\begin{array}{r} 0 \ 0 \ 0 \ 0 \overset{1}{\underset{1}{\text{1}}} \ 1 \ 1 \overset{1}{\underset{1}{\text{0}}} \ 1 \\ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \end{array}$$

The sum gives us $0000\ 1101_{2C} + 1110\ 1001_{2C} = 1111\ 0110_{2C}$.

The decimal representation can be found by noting the leftmost bit is 1, so the value is negative. Invert and add one: $-0000\ 1010_2 = -10_{10}$.

Alternatively, the decimal representation can be found by using the shortcut conversion tool.

$$\begin{array}{r} 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \\ -2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0 \end{array}$$

Add each place.

$$\begin{array}{rcl} 1 * -2^7 & = & -128 \\ 1 * 2^6 & = & 64 \\ 1 * 2^5 & = & 32 \\ 1 * 2^4 & = & 16 \\ 0 * 2^3 & = & 0 \\ 1 * 2^2 & = & 4 \\ 1 * 2^1 & = & 2 \\ 0 * 2^0 & = & 0 \\ \hline & & -10 \end{array}$$

Again, the sum of these is -10_{10} .

5. *Problem:* Show how the computer performs the addition $-20_{10} + 23_{10}$ in eight bit Two's Complement.

Solution: First, convert both values into binary and pad bits as needed. $-20_{10} = -0001\ 0100_2$ and $23_{10} = 0001\ 0111_2$. Next, convert these values into Two's Complement.



The positive value needs no alteration, and the negative value undergoes “invert and add one”.

The value 0001 0100 inverts into 1110 1011. Add one:

$$\begin{array}{r}
 & & & & & 1 & 1 \\
 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
 & & & & & & & 1 \\
 \hline
 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
 \end{array}$$

This gives us an expression of $1110\ 1100_{2C} + 0001\ 0111_{2C}$

$$\begin{array}{r}
 & & & & & 1 & 1 & 1 \\
 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
 \hline
 10 & 0 & 0 & 0 & 0 & 0 & 1 & 1
 \end{array}$$

The overflow bit falls away, and in the case of Two’s Complement, does not indicate an error condition. This gives a result of $1110\ 1100_{2C} + 0001\ 0111_{2C} = 0000\ 0011_{2C}$.

Converting back into decimal, we note the result value is positive, so no transformation is required. $0000\ 0011_{2C} = 0000\ 0011_2 = 3_{10}$.

6. *Problem:* What happens when the computer attempts to perform the addition $-100_{10} + (-80_{10})$ in eight bit Two’s Complement? How can the error be detected?

Solution: First, convert both values into binary and pad bits as needed. $-100_{10} = -0110\ 0100_2$ and $-80_{10} = -0101\ 0000_2$. Both numbers are negative and so we must invert and add one to convert into Two’s Complement. This gives $-100_{10} = 1001\ 1100_{2C}$ and $-80_{10} = 1011\ 0000_{2C}$. The latter shares a right hand block of 0s with its original due to a substantial repeated carry from the add-one.

$$\begin{array}{r}
 & & & & & 1 & 1 \\
 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 \hline
 10 & 1 & 0 & 0 & 1 & 1 & 0 & 0
 \end{array}$$

The overflow bit is set, but this does not indicate an error condition in Two’s Complement. The result, with the extra bit fallen away, is: $1001\ 1100_{2C} + 1011\ 0000_{2C} = 0100\ 1100_{2C}$. This result can still be shown to be in error by sign checking. Sign checking indicates that a Two’s Complement addition of two numbers with the same sign must have a result with that same sign. In this case, both numbers were negative, so the result must be negative. This result has a leftmost bit of zero, which indicates it is positive. Therefore, the computation is in error.

This can be confirmed conceptually by noting that $-100_{10} + (-80_{10}) = -180_{10}$, which is outside the range of eight bit Two’s Complement.



7. *Problem:* Convert the number 354_{10} into binary via BCD.

Solution: First, convert into BCD. BCD allocates a 4-bit block for each decimal digit. Thus, $354_{10} = 0011\ 0101\ 0100_{BCD}$. The three place values, from left to right, are $100_{10} = 1100100_2$, $10_{10} = 1010_2$, and $1_{10} = 1_2$. Multiply each 4-bit block by its place value. The multiplications are $0011_2 * 1100100_2$, $0101_2 * 1010_2$, and $0100_2 * 1_2$.

First, solve $0011_2 * 1100100_2$. Using the commutative law, we can solve this as $1100100_2 * 0011_2$. There are 1 bits in the first and second place, giving shifts of $1100100_2 \ll 0_{10} = 1100100_2$, and $1100100_2 \ll 1_{10} = 11001000_2$ (note that there is no fixed width here, so we don't drop away any digits off the left).

Add these results together:

$$\begin{array}{r} & 1 \\ & 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \\ \hline 10 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \end{array}$$

Thus, $0011_2 * 1100100_2 = 100101100_2$.

Next, solve $0101_2 * 1010_2$. There are 1 bits in the second and fourth place, giving shifts of $0101_2 \ll 1_{10} = 01010_2$ and $0101_2 \ll 3_{10} = 0101000_2$

$$\begin{array}{r} & 1 \\ & 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \\ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \\ \hline 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \end{array}$$

Thus, $0101_2 * 1010_2 = 110010_2$ (note that the leftmost zero bits have no significance and can be dropped when working with the individual blocks).

Finally, solve $0100_2 * 1_2$. Anything multiplied by 1 is itself, so this remains 0100_2 .

Add all the intermediate results together.

$$\begin{array}{r} & 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \\ & 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \\ & \quad 1 \ 0 \ 0 \\ \hline 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \end{array}$$

Thus, $354_{10} = 0011\ 0101\ 0100_{BCD} = 1\ 0110\ 0010_2$. Note that no bit width was specified, so expanding to use nine bits is not a problem.



8. *Problem:* Convert the number 287_{10} into BCD via binary.

Solution: First, convert into binary: $287_{10} = 1\ 0001\ 1111_2$. To convert this binary number into BCD, apply the double dabble algorithm. Start by configuring the space. How many BCD blocks are needed? Apply the formula $\lceil \log_{10} 2^n \rceil * 4$ to determine this. We find that $2^9 = 512_{10}$, and $\lceil \log_{10} 512_{10} \rceil = 3_{10}$. Three BCD blocks will require 12 bits total. Attach the original number to the right side of the space: 0000 0000 0000 100011111.

	BCD Blocks			Input	Operation
1.	0000	0000	0000	100011111	Start
1.	0000	0000	0000		none > 4
3.	0000	0000	0001	00011111	Shifted Left
4.	0	0	1		none > 4
5.	0000	0000	0010	0011111	Shifted Left
6.	0	0	2		none > 4
7.	0000	0000	0100	011111	Shifted Left
8.	0	0	4		none > 4
9.	0000	0000	1000	11111	Shifted Left
10.	0	0	8		Rightmost block exceeds four
11.	0000	0000	1011	11111	Added 11_2 to rightmost block
12.	0000	0001	0111	1111	Shifted Left
13.	0	1	7		Rightmost block exceeds four
14.	0000	0001	1010	1111	Added 11_2 to rightmost block
15.	0000	0011	0101	111	Shifted Left
16.	0	3	5		Rightmost block exceeds four
17.	0000	0011	1000	111	Added 11_2 to rightmost block
18.	0000	0111	0001	11	Shifted Left
19.	0	7	1		Middle block exceeds four
20.	0000	1010	0001	11	Added 11_2 to middle block
21.	0001	0100	0011	1	Shifted Left
22.	1	4	3		none > 4
23.	0010	1000	0111		Shifted Left, final result

Thus, $287_{10} = 1\ 0001\ 1111_2 = 0010\ 1000\ 0111_{BCD}$.



A.11 Floating Point Numbers

Exercises found in Chapter 11 on page 122.

1. *Problem:* Convert 14.25_{10} into 16-bit binary floating point.

Solution: First convert into binary. The whole number and fractional portion are converted separately, with $14_{10} = 1110_2$ and $0.25_{10} = 0.01_2$. The fractional portion can be seen without going through a lot of work by realizing that the first place right of the decimal is a half, followed by a quarter, and so on. So the entire number $14.25_{10} = 1110.01_2$.

Next, represent the binary number in normalized scientific notation: $1.11001 * 2^3$. The leading one (left of the decimal point) will fall away as an implicit 1. The remaining significand is 11001.

The exponent must be adjusted by the exponent bias. For 16-bit floating point, the exponent bias is 15. $3_{10} + 15_{10} = 18_{10} = 10010_2$.

The number is positive, so the sign bit will be 0. Padding out the significand to fill the required bits, the final value is 0100 1011 0010 0000 = $4B20_{16}$.

2. *Problem:* Convert -2.67_{10} into 16-bit binary floating point.

Solution: First convert into binary. The whole number and fractional portion are converted separately, with $2_{10} = 10_2$. How about 0.67_{10} ? Convert using the repeated multiplication technique:

$$\begin{array}{rcl}
 0.67 * 2 & = & 1.34 \text{ (digit 1)} \\
 0.34 * 2 & = & 0.68 \text{ (digit 0)} \\
 0.68 * 2 & = & 1.36 \text{ (digit 1)} \\
 0.36 * 2 & = & 0.72 \text{ (digit 0)} \\
 0.72 * 2 & = & 1.44 \text{ (digit 1)} \\
 0.44 * 2 & = & 0.88 \text{ (digit 0)} \\
 0.88 * 2 & = & 1.76 \text{ (digit 1)} \\
 0.76 * 2 & = & 1.52 \text{ (digit 1)} \\
 0.52 * 2 & = & 1.04 \text{ (digit 1)} \\
 0.04 * 2 & = & 0.08 \text{ (digit 0)}
 \end{array}$$

We can stop at this point, although the digits continue, because we know that 16-bit floating point only has ten significant bits.

So the number we have is $10.1010101110\cdots_2$. Rewrite in normalized scientific notation $1.01010101110\cdots * 2^1$. The leftmost one bit drops away (implicit 1), and we have ten bits to store the significand in. Count out the first ten bits: 0101010111. This becomes the significand.



The exponent, 1, as adjusted in the usual way by adding the exponent bias: $1_{10} + 15_{10} = 16_{10} = 10000_2$.

Finally, the number is negative, so apply a sign bit of 1, and attach the exponent and significand. The final value is $\underline{1100\ 0001\ 0101\ 0111} = C157_{16}$.

3. *Problem:* Convert $\frac{1}{3}$ (base 10) into 16-bit binary floating point.

Solution: Convert into binary. There is no whole number portion, so the fractional portion will be converted. We know in advance that rounding will occur, so we could convert the fraction into a decimal of a dozen places or so (it would be more than ten, as we need ten bits after the first 1). It is also possible to continue with the fraction representation.

$$\begin{aligned}\frac{1}{3} * 2 &= \frac{2}{3} \text{ (digit 0)} \\ \frac{2}{3} * 2 &= 1\frac{1}{3} \text{ (digit 1)} \\ \frac{1}{3} * 2 &= \frac{2}{3} \text{ (digit 0)} \\ \frac{2}{3} * 2 &= 1\frac{1}{3} \text{ (digit 1)} \\ \dots \text{ The pattern continues.}\end{aligned}$$

The number, in binary, is $0.010101\dots$. Shifting this into scientific notation, we find $1.010101\dots * 2^{-2}$. The leftmost one is implicit, and falls away. We need ten bits to fill in the significand, so repeat the pattern until ten bits are filled: 0101010101 .

The exponent also needs to be biased: $-2_{10} + 15_{10} = 13_{10} = 1101_2$. Remember to pad the exponent, if needed, so that it occupies the right number of bits. In this case (16 bit floating point), the exponent should occupy five bits: 01101 .

The final value is $\underline{0011\ 0101\ 0101\ 0101} = 3555_{16}$.

4. *Problem:* Convert the 16-bit floating point number 9876_{16} into decimal.

Solution: First, break the hexadecimal into binary: $9876_{16} = 1001\ 1000\ 0111\ 0110_2$. Locate the sign bit, exponent, and significand based on the known sizes of each of these. The sign bit is leftmost, followed by five exponent bits (as defined for 16-bit floating point), and the remainder are significand bits: $\underline{1001\ 1000\ 0111\ 0110}$.

The biased exponent is adjusted to yield $00110_2 = 6_{10} - 15_{10} = -9_{10}$. The sign bit is negative, so at this point we have $-1.0001110110 * 2^{-9} = -0.0000000010001110110_2$. Note the extra (implicit) one bit that was placed back on in the scientific notation. It is important not to forget the implicit one.

To convert -0.0000000010001110110_2 into decimal, start at the 2^{-9} place.



$$\begin{array}{cccccccccccccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 2^{-9} & 2^{-10} & 2^{-11} & 2^{-12} & 2^{-13} & 2^{-14} & 2^{-15} & 2^{-16} & 2^{-17} & 2^{-18} & 2^{-19} \end{array}$$

Add up each place.

$$\begin{array}{rcl}
 1 * 2^{-9} & = & 0.001953125 \\
 0 * 2^{-10} & = & 0 \\
 0 * 2^{-11} & = & 0 \\
 0 * 2^{-12} & = & 0 \\
 1 * 2^{-13} & = & 0.0001220703125 \\
 1 * 2^{-14} & = & 0.00006103515625 \\
 1 * 2^{-15} & = & 0.000030517578125 \\
 0 * 2^{-16} & = & 0 \\
 1 * 2^{-17} & = & 0.00000762939453125 \\
 1 * 2^{-18} & = & 0.000003814697265625 \\
 0 * 2^{-19} & = & 0 \\
 \hline
 & & 0.002178192138671875
 \end{array}$$

Attach the whole number portion (none in this case), and apply the sign bit. The final value is $-0.002178192138671875_{10}$.

5. *Problem:* Convert the 16-bit floating point number $7BFF_{16}$ into decimal. What, if anything, is special about this value?

Solution: First, break the hexadecimal into binary: $7BFF_{16} = 0111\ 1011\ 1111\ 1111_2$. Locate the sign bit, exponent, and significand based on the known sizes of each of these. The sign bit is leftmost, followed by five exponent bits (as defined for 16-bit floating point), and the remainder are significand bits: 0111 1011 1111 1111₂.

The biased exponent is adjusted to yield $11110_2 = 30_{10} - 15_{10} = 15_{10}$. The sign bit is positive, so at this point we have $1.1111111111 * 2^{15} = 1111\ 1111\ 1110\ 0000_2$. There is no fractional component, so this number is a whole number. Performing a normal binary to decimal conversion yields 65504_{10} .

This value is special because it is the largest actual number representable in 16-bit binary floating point. If the exponent bits were all 1s, that would indicate a special case condition, so the exponent is the largest it can be and still be a valid exponent. The significand likewise is the largest significand possible. Together, they form the largest value that can be represented.

6. *Problem:* What is the smallest positive integer which *cannot* be represented in 16-bit binary floating point?

Solution: We have previously seen that 65504_{10} is the largest number that can be represented in 16-bit binary floating point. However, the next number down, found by decreasing the least significant bit of the significand, is a substantial step: 0111 1011 1111 1110₂. This



becomes $1.111111110 * 2^{15} = 1111\ 1111\ 1100\ 0000_2 = 65472_{10}$. Thus, it is not possible to represent numbers between 65472_{10} and 65504_{10} in 16-bit binary floating point.

For example, 65473_{10} cannot be represented in 16-bit binary floating point.

To systematically find the smallest such non-representable number, work backwards from the binary result. Notice that the exponent can force a group of zeros on the right edge of the number, which is the source of numbers being skipped. Move the exponent so that the very first skip occurs: $1111\ 1111\ 1110_2$. If we moved the number one more space to the right, there would be no skip, as the significand fills in all the remaining bits. The number $1111\ 1111\ 1110_2$ is representable, but $1111\ 1111\ 111\underline{1}_2$ is not, as there is not enough space in the significand. Thus, 4095_{10} cannot be represented in 16-bit binary floating point.

Could there be a smaller positive integer still? Instead of filling the significand with 1s, we could fill it with 0s. We need only to ensure that the leftmost 1 remains, otherwise the exponent will simply adjust to consume the empty space. This gives $1000\ 0000\ 0001_2$. Think about it: The leftmost one becomes implicit, and ten bits remain in the significand. This doesn't allow for the final, rightmost 1 to be represented. The value $1000\ 0000\ 0001_2 = 2049_{10}$, the smallest positive integer not representable in 16-bit binary floating point.

Check: Represent 2049_{10} in 16-bit binary floating point. Start with the binary value $1000\ 0000\ 0001_2$ and apply scientific notation $1.000\ 0000\ 0001 * 2^{11}$. Bias the exponent $11_{10} + 15_{10} = 26_{10} = 11010_2$. Altogether, with ten bits of significand (the leftmost one being implicit), the number is 0110 1000 0000 0000. The rightmost 1 bit falls away due to insufficient representation space. Converting back, 0110 1000 0000 0000 becomes $1 * 2^{11} = 2048_{10}$. So the attempt to represent 2049_{10} failed.

Integers smaller than 2048_{10} can be represented because they will have a smaller exponent, meaning that the whole number place will be one to the left, and the bit that fell away (in this case) would be to the right of the decimal place, and not needed for integers.

7. *Problem:* In 16-bit binary floating point, how many numbers can be represented between 2_{10} and 3_{10} , inclusively?

Solution: First, convert each of these values into binary scientific notation: $2_{10} = 10_2 = 1.0 * 2^1$, and $3_{10} = 11_2 = 1.1 * 2^1$. Notice the exponent is the same, so any value which falls between these two will have the same exponent (although some values outside of that range will share the exponent, such as 3.5_{10}). Accounting for the implicit 1 left of the decimal place, the range for the significand will be 00 0000 0000 to 10 0000 0000 (being the significands for 2_{10} and 3_{10} , respectively). It is acceptable to just compare the significands, because the exponent is the same.

Based on the value for 2_{10} , the nine bits remain, which can take on any value: $2^9 = 512$ possibilities (for 00 0000 0000 through 01 1111 1111). The value of 3_{10} is the upper limit, and any change to its significand would give a number greater than three. Thus, there are a total of 513 representable numbers between 2_{10} and 3_{10} , inclusively.

8. *Problem:* In 16-bit binary floating point, how many numbers can be represented between 0_{10} and 1_{10} , inclusively?

Solution: This problem is somewhat more intense, because a variety of exponents are possible. The value $1_{10} = 1_2 = 1 * 2^0$. Bias the exponent $0_{10} + 15_{10} = 15_{10} = 01111_2$. Any exponent higher than this value will produce a larger number. With this exponent, only a significand of all zeros will produce the value of 1_{10} ; any other significand will produce a larger number.

So, we are looking at exponents smaller than 01111_2 . The smallest exponent is 00000_2 , a special case which indicates zero (to be included) and subnormal numbers (also to be included). So for our purposes, there is no unusual consideration for the zero exponent.

For each exponent, there exists ten significand bits, with $2^{10} = 1024$ possibilities. The exponent range being considered is 00000_2 through 01110_2 , fifteen unique exponents. $1024_{10} * 15_{10} = 15360_{10}$. This includes zero and all values up to but not including 1_{10} . Adding the value of one gives the total count of representable values as 15361.

Compare this answer to the previous problem: in both cases, there was a range of 1. Yet, the count of representable values in that range varied tremendously. Looking at the prior problems, with large numbers it could be difficult even to represent a whole number value, much less the fractions in between. In general, binary floating point numbers have the most representation capability near zero, and the weakest representation capability in numbers further from zero.

A.12 Unicode and ASCII

Exercises found in Chapter 12 on page 129.

1. *Problem:* Show how the phrase “Tall tree” would appear in UTF-8.

Solution: Each symbol in the phrase is covered by the original 7-bit ASCII, so one byte will be used for each symbol. Be careful to note the difference between uppercase and lowercase, and account for the space symbol. Each symbol can be identified in the ASCII chart shown earlier. The rows are most significant bits, and the columns are the least significant bits. The capital T, for example, is found in row 5_{16} and column 4_{16} . Thus the T is encoded as 54_{16} . The remaining symbols are encoded likewise. The space is found at row 2_{16} and column 0_{16} , so its encoding is 20_{16} .

The entire phrase is encoded as $54\ 61\ 6C\ 6C\ 20\ 74\ 72\ 65\ 65_{16}$.

2. *Problem:* Find the UTF-8 encoding for the Unicode code point 2399_{10} .

Solution: First, find the Unicode code point in hexadecimal. This is needed because all the encoding rules are based on the hexadecimal ranges. We find that $2399_{10} = 95F_{16}$. Check the encoding rules for U+095F to find how it should be encoded. This code point falls into the third category. Once converted into binary, the encoding $1110\ abcd\ 10ef\ ghij\ 10kl\ mnop$ will be used.

Converting $95F_{16}$ into binary gives $1001\ 0101\ 1111_2$. Be careful! We need to ensure we have the same number of bits that the code point listing has. For this encoding rule, the code point range include 16 bits. Thus, we need to first pad out our number (by adding 0s on the left) to include 16 bits. The padded value is $0000\ 1001\ 0101\ 1111_2$

Substitute the bits into the pattern given (recall that *a* refers to the leftmost bit, and so on). This gives $1110\ \underline{0000}\ \underline{1010}\ \underline{0101}\ \underline{1001}\ \underline{1111}_2 = E0\ A5\ 9F_{16}$.

The Unicode code point 2399_{10} (U+095F) is encoded in UTF-8 as $E0\ A5\ 9F_{16}$.

3. *Problem:* Find the Unicode code point associated with the UTF-8 encoding $D7\ 91_{16}$.

Solution: First, identify which encoding pattern is in use. The sequence $D7\ 91_{16} = 1101\ 0111\ 1001\ 0001_2$. The first three bits are 110, which matches a two byte second row pattern of $110a\ bcde\ 10fg\ hijk$. Match the UTF-8 encoded bits against the pattern, determining which bits participate in the code point. The code point bits are underlined: $\underline{1101}\ \underline{0111}\ \underline{1001}\ \underline{0001}_2$. Extract the code point bits, giving $101\ 1101\ 0001_2 = 5D1_{16}$.

The UTF-8 sequence $D7\ 91_{16}$ is associated with the Unicode code point U+05D1.

4. *Problem:* A certain document contains 1,500 English letters, 350 standard punctuation marks (including space), and 50 symbols in Arabic. Arabic symbols have Unicode code



points from U+0600 through U+06FF. Determine how many bytes of disk space this document will take to store using UCS-2 compared to UTF-8. (Assume no overhead for formatting; only the encoded characters will be stored).

Solution: All English letters and standard punctuation are found on the 7-bit ASCII table, and so can be encoded with one byte in UTF-8. Unicode code points in the U+06xx range are second row (two byte) UTF-8 encodings. Thus, for UTF-8, this document will require $1500 + 350 + 50 * 2 = 1950$ bytes.

In UCS-2, all code points are encoded with two bytes. Thus, for UCS-2, this document will require $(1500 + 350 + 50) * 2 = 3800$ bytes. Due to the prevalence of English letters, the UCS-2 will require almost double the size compared to the UTF-8 encoding.

5. *Problem:* The “Heart Sutra” is a famous Buddhist text. It is very short, at only 260 Chinese symbols. Chinese symbols have Unicode code points from U+4E00 through U+9FCF. Determine how many bytes of disk space this document will take to store using UCS-2 compared to UTF-8. (Assume no overhead for formatting; only the encoded characters will be stored).

Solution: First, determine how many bytes per symbol are needed for UTF-8. The code points in the range specified all fall into the third block (U+0800 to U+FFFF), requiring three bytes per symbol. Thus, the document will require $260 * 3 = 780$ bytes to encode in UTF-8.

Recall that UCS-2 uses exactly two bytes for all Unicode code points. Thus, UCS-2 will require $260 * 2 = 520$ bytes to encode this document. Perhaps surprisingly, UCS-2 is more efficient than UTF-8. Some countries have objected to the UTF-8 encoding on the grounds that it favors English and European languages at the expense of Asian languages.

6. *Problem:* One English translation of the “Heart Sutra” contains 1,357 English symbols. Will this document (again, assuming no overhead), encoded in UTF-8, take more or less space than the Chinese version, also encoded in UTF-8?

Solution: Encoding English letters in UTF-8 requires one byte per symbol. Thus, this document will require 1,357 bytes in UTF-8. The Chinese version required only 780 bytes in UTF-8. The English translation, although it uses one byte instead of three bytes per symbol, still takes almost twice as much space to encode due to higher symbol count.

7. *Problem:* The word “hello” in Chinese is rendered as 你好. In UTF-8, these two symbols are encoded as E4 BD A0 E5 A5 BD₁₆. Find the Unicode code points for these symbols.

Solution: More than one symbol is involved, so to determine how many bytes to read, we must consult the leftmost bits. The first four bits $E_{16} = 1110_2$. A pattern beginning with 1110 uses three bytes, so we will take the first three bytes ($E4\ BD\ A0$) for the first symbol and leave the rest for the remaining symbol(s).



First, translate the hexadecimal representation into binary $E4\ BD\ A0_{16} = 1110\ 0100\ 1011\ 1101\ 1010\ 0000_2$. A sequence beginning with 1110 matches the third row, three byte pattern 1110 *abcd10ef ghij 10kl mnop*. Mark which bits in the UTF-8 sequence are data in the pattern: 1110 0100 1011 1101 1010 0000₂. Extract the bits based on the pattern: 0100 1111 0110 0000₂ = 4F60₁₆. The first symbol has the Unicode code point U+4F60.

Moving on, the next four bits $E_{16} = 1110_2$ also indicate a third tow, three byte pattern. The remaining three bytes ($E5\ A5\ BD_{16}$) must represent a single symbol.

Translate the hexadecimal representation into binary $E5\ A5\ BD_{16} = 1110\ 0101\ 1010\ 0101\ 1011\ 1101_2$. A sequence beginning with 1110 matches the third row, three byte pattern 1110 *abcd10ef ghij 10kl mnop*. Mark which bits in the UTF-8 sequence are data in the pattern: 1110 0101 1010 0101 1011 1101₂. Extract the bits based on the pattern: 0101 1001 0111 1101₂ = 597D₁₆. The second symbol has the Unicode code point U+597D.

The two symbols have the Unicode code points U+4F60 and U+597D respectively.

8. *Problem:* Translate the 8-bit Two's Complement number $1101\ 0101_{2C}$ into UTF-8 for display.

Solution: In order to display this value, the digits associated with must be determined. First, given that the value is Two's Complement, determine if it is positive or negative. The leading 1 indicates this is a negative number. Negative numbers are displayed starting with a dash (minus sign), which, according to the ASCII table, is symbol U+002D (recall that the 7-bit ASCII symbols have the same code points as Unicode).

Next, convert the number itself. Find the absolute value by inverting (0010 1010) and adding one, yielding 0010 1011₂. This value must then be converted into BCD. An algorithm such as the double dabble could be used to find that 0010 1011₂ = 43₁₀ = 0100 0011_{BCD}.

Looking up the digit zero, we find it has the Unicode code point U+0030. The digits are in sequential order, so for each digit, simply add the BCD representation to the base code point of U+0030 to find the code point for the digit. The leftmost digit is 30₁₆ + 4₁₆ = 34₁₆. The rightmost digit is 30₁₆ + 3₁₆ = 33₁₆. Thus, the Unicode code points for the digits are U+0034 followed by U+0033.

All together, including the negative sign, three symbols are needed: U+002D, U+0034, U+0033.

Each of these code points must now be encoded into UTF-8. Fortunately, each of them follows the first row, one byte encoding. Thus, the UTF-8 encoding of this sequence is just 2D 34 33₁₆.



A.13 Images and Color

Exercises found in Chapter 13 on page 144.

1. *Problem:* Describe the colors represented by the following RGB codes.

- (a) #4682B4
- (b) #C71585
- (c) #A52A29
- (d) #807E82
- (e) #98FB99

Solution:

- (a) #4682B4 This color is dominated by the blue component $B4_{16}$, with a secondary of green. Green and blue together form cyan, but this color is slightly darker and has a significantly increased red component. Thus, we could describe it as a “grayish blue with a hint of green”.
- (b) #C71585 This color is dominated by the red component, with a secondary of blue. There is minimal green in this color. If red and blue were evenly mixed, the color would be purple; but the red dominates. Therefore, the color is more of a red-violet.
- (c) #A52A29 This color is dominated by the red component, with an almost even mix of blue and green. The blue and green temper and dull the red, making it more orange or brown.
- (d) #807E82 This color consists of three almost equal components ($7E_{16}$ is only 2 away from 80_{16}). If the red, green, and blue components are nearly equal, the color is a shade of gray. Given that the values are in the middle range of spectrum, this gray is a middle gray (halfway between white and black).
- (e) #98FB99 This color is strongly green, with a substantial amount of red and blue as well; the high levels of all three components suggest a bright color. The red and blue are about equal, so the color will remain centered on green. A light, pale green, is the likely result.

2. *Problem:* Describe the colors represented by the following HSV values.

- (a) 260 degrees, 35% saturation, 94% value
- (b) 134 degrees, 65% saturation, 41% value
- (c) 46 degrees, 10% saturation, 95% value
- (d) 321 degrees, 10% saturation, 9% value



- (e) 321 degrees, 90% saturation, 9% value

Solution:

- (a) 260 degrees, 35% saturation, 94% value

The 260 degree mark on the hue wheel is predominately blue with a good amount of red as well. The very high value indicates a bright color, and the somewhat low saturation indicates a washed out color. Blue-red combines, and with the saturation and value, we expect a bright, washed out purple.

- (b) 134 degrees, 65% saturation, 41% value

The 134 degree mark is primarily green, with a secondary of blue. The high saturation indicates a vivid color, but the somewhat low value indicates a darker color. A darker, vivid green (with a little blue) could best be described as forest green.

- (c) 46 degrees, 10% saturation, 95% value

The 46 degree mark indicates a red domination with green as well. Red together with green forms yellow. The low saturation indicates a very washed out color; combined with the high value, this color will be nearly white. An off-white, slightly yellowish will result.

- (d) 321 degrees, 10% saturation, 9% value

The 321 degree mark is red with some blue. Note the very low saturation indicates a washed out color, and the very low value indicates a very dark color. Together, the low saturation and low value indicate this color will be (regardless of hue), essentially, black.

- (e) 321 degrees, 90% saturation, 9% value

Like the previous example, except the high saturation indicates a dark, vivid color. Red with blue forms a shade of purple, so with the low value, we expect a very dark purple.

3. *Problem:* Convert the HSV color with hue of 200 degrees, a saturation of 50%, and a value of 85% to RGB.

Solution: The dominant RGB component, based on the hue categories, is blue, with a decreasing secondary component of green. The dominant RGB component (blue) will be assigned the value, so $B = 0.85$, assuming a 0 through 1 scale for RGB to start with.

Next, we must calculate how far (as a percentage) through the current hue category the color is. This can be found by calculating $F = \frac{H}{60} - \left\lfloor \frac{H}{60} \right\rfloor = \frac{200}{60} - \left\lfloor \frac{200}{60} \right\rfloor = 3\frac{1}{3} - 3 = \frac{1}{3}$.

The secondary RGB component, identified as green, is calculated using the decreasing formula $V - VSF$. Therefore, $G = V - VSF = 0.85 - 0.85 * 0.5 * 0.33 = 0.85 - 0.14 = 0.71$.

Finally, the remaining component (red) is defined as $R = V - VS = 0.85 - 0.85 * 0.5 = 0.85 - 0.43 = 0.42$.



The RGB tuple $(0.42, 0.71, 0.85)$ can be converted into RGB code by first multiplying each component by 255, and then rendering them as hexadecimal. The multiplication yields $(107_{10}, 181_{10}, 217_{10})$. We convert these values to hexadecimal, yielding $(6B_{16}, B5_{16}, D9_{16})$ which in turns gives an RGB code of $\#6BB5D9$.

4. *Problem:* Convert the RGB color $\#274F07$ into HSV.

Solution: First, we must convert RGB into a tuple of values 0 through 1. The RGB begins as a hexadecimal tuple $(27_{16}, 4F_{16}, 07_{16})$. Next, we convert these into decimal, yielding $(39_{10}, 79_{10}, 7_{10})$, and then divide each by 255 to find a 0 through 1 range. The final RGB tuple is $(0.153, 0.31, 0.027)$.

The value is defined as the dominant RGB component, or $V = \max(R, G, B)$. The dominant component in this color is green, so $V = 0.31$.

Next, we define the intermediate value $D = V - \min(R, G, B)$ (in other words, the difference between the largest and smallest RGB components). In this case, $D = 0.31 - 0.027 = 0.283$. With D , the saturation can be defined as $S = \frac{D}{V} = \frac{0.283}{0.31} = 0.913$.

We can determine the hue by first noting that green is dominant in this color, so the green dominant hue formula will be applied: $H = 60 * \left(2 + \frac{B - R}{D}\right) = 60 * \left(2 + \frac{0.027 - 0.153}{0.283}\right) = 60 * (2 + -0.445) = 60 * 1.555 = 93.3$.

The final HSV values are 93 degrees hue, 91% saturation, and 31% value.

5. *Problem:* A large uncompressed, unpacked black and white (1 bit) raster image requires about 100,000 bytes. If the image is instead stored as uncompressed, *packed* black and white (1 bit), about how much space will be required?

Solution: Unpacked images store not more than one pixel per byte. Given that a byte is 8 bits, a packed 1 bit image could store eight times more data per byte than an unpacked 1 bit image. Thus, the packed version of the above image can be stored in about $\frac{100000}{8} = 12500$ bytes.

6. *Problem:* An uncompressed 24-bit color raster image requires about 200,000 bytes. If an 8-bit alpha channel is added to the image, about how much space will the new image require?

Solution: If each pixel is using 24 bits, that means each pixel uses 3 bytes. We can calculate the number of pixels as about $\frac{200000}{3} = 66,666$. Now, by adding an 8 bit alpha channel, each pixel will use a total of 4 bytes. Thus, the approximate space required is now $66,666 * 4 = 266,664$.



Alternative Solution: Apply the formula $3 + \frac{nwh}{8}$. The original image has 24-bit color, so $3 + \frac{24wh}{8} = 200,000$. The term wh will equal the total number of pixels in the image, so solve for wh . This yields approximately $wh = 66,666$. Next, with an additional 8 bit alpha channel, the image size will be $3 + \frac{(24+8)(66,666)}{8} = 266,667$ bytes approximately.

7. *Problem:* An opaque background of color #3409AB is overlaid with the translucent ARGB color #0F4A7A99. What is the resulting RGB display color?

Solution: The background color is opaque (has no alpha channel, or an alpha channel of 1), so apply the simplified formulas. First, convert the RGB codes of the top (foreground) color into 0 through 1 ranges. The foreground color has red $4A_{16} = 74_{10}/255_{10} = 0.29$; green $7A_{16} = 122_{10}/255_{10} = 0.48$; blue $99_{16} = 153_{10}/255_{10} = 0.6$; alpha $0F_{16} = 15_{10}/255_{10} = 0.06$. Thus, we can define $(R_0, G_0, B_0, A_0) = (0.29, 0.48, 0.6, 0.06)$.

Likewise, the background color can be converted into the 0 through 1 ranges. The background color has red $34_{16} = 52_{10}/255_{10} = 0.20$; green $09_{16} = 9_{10}/255_{10} = 0.04$; blue $AB_{16} = 171_{10}/255_{10} = 0.67$. Thus, we can define $(R_1, G_1, B_1) = (0.20, 0.04, 0.67)$.

We can calculate the resulting color using the simplified formulas $R_2 = A_0 R_0 + (1 - A_0) R_1$, and likewise for G_2 and B_2 . The resulting color will have no alpha channel (or an alpha of 1), since one of the source colors is opaque.

$$\text{Find } R_2 = A_0 R_0 + (1 - A_0) R_1 = 0.06 * 0.29 + (1 - 0.06) * 0.20 = 0.02 + 0.19 = 0.21.$$

$$\text{Next, find } G_2 = A_0 G_0 + (1 - A_0) G_1 = 0.06 * 0.48 + (1 - 0.06) * 0.04 = 0.03 + 0.04 = 0.07.$$

$$\text{Finally, find } B_2 = A_0 B_0 + (1 - A_0) B_1 = 0.06 * 0.6 + (1 - 0.06) * 0.67 = 0.04 + 0.63 = 0.67.$$

Thus, the resulting color is $(R_2, G_2, B_2) = (0.21, 0.07, 0.67)$.

If desired, this color can be converted back into RGB code by multiplying each value by 255 and converting to hexadecimal. Thus, $(0.21, 0.07, 0.67) * 255 = (54_{10}, 18_{10}, 171_{10})$. Converting each to hexadecimal yields #3612AB.

Note if this conversion is performed by computer the result will be slightly different due to rounding. For example, when this overlap is performed in a popular image editing program, the resulting color is #3510AA.

8. *Problem:* An ARGB color #B1A00591 is placed on top of another ARGB color #55667788. The two colors are alpha blended; what is the resulting ARGB color?

Solution: First, convert each color into percentage values for each component. The first mentioned color is placed on top, and the second on bottom (due to the “on top of” phrase). Therefore, the first (top) color will be defined as (R_0, G_0, B_0, A_0) .

The top color has red $66_{16} = 102_{10}/255_{10} = 0.4$; green $77_{16} = 119_{10}/255_{10} = 0.47$; blue $88_{16} = 136_{10}/255_{10} = 0.53$; alpha $55_{16} = 85_{10}/255_{10} = 0.33$. The bottom color has red $A0_{16} = 160_{10}/255_{10} = 0.63$; green $05_{16} = 5_{10}/255_{10} = 0.02$; blue $91_{16} = 145_{10}/255_{10} = 0.57$; alpha $B1_{16} = 177_{10}/255_{10} = 0.69$. These values create the tuples for each color. The top



color is $(R_0, G_0, B_0, A_0) = (0.4, 0.47, 0.53, 0.33)$ and the bottom color is $(R_1, G_1, B_1, A_1) = (0.63, 0.02, 0.57, 0.69)$.

The blended alpha value, A_2 , can be calculated with the formula $A_2 = A_0 + (1 - A_0)A_1$. Let $A_2 = 0.33 + (1 - 0.33)0.69 = 0.33 + 0.46 = 0.79$.

For each primary component (red, green, and blue), the formula $R_2 = \frac{A_0 R_0 + (1 - A_0)(A_1 R_1)}{A_2}$ can be applied.

$$\text{Let } R_2 = \frac{0.33 * 0.4 + (1 - 0.33)(0.69 * 0.63)}{0.79} = \frac{0.13 + 0.29}{0.79} = \frac{0.42}{0.79} = 0.53.$$

$$\text{Next, } G_2 = \frac{A_0 G_0 + (1 - A_0)(A_1 G_1)}{A_2} = \frac{0.33 * 0.47 + (1 - 0.33)(0.69 * 0.02)}{0.79} = \frac{0.16 + 0.01}{0.79} = \frac{0.17}{0.79} = 0.22.$$

$$\text{Finally, } B_2 = \frac{A_0 B_0 + (1 - A_0)(A_1 B_1)}{A_2} = \frac{0.33 * 0.53 + (1 - 0.33)(0.69 * 0.57)}{0.79} = \frac{0.17 + 0.26}{0.79} = \frac{0.43}{0.79} = 0.54.$$

Thus, the new color $(R_2, G_2, B_2, A_2) = (0.53, 0.22, 0.54, 0.79)$. This color can be converted into an ARGB code by multiplying each component by 255 and converting to hex. Thus, $(0.53, 0.22, 0.54, 0.79) * 255 = (135_{10}, 56_{10}, 138_{10}, 201_{10}) = (87_{16}, 38_{16}, 8A_{16}, C9_{16})$. Note that in ARGB code the alpha value goes first, so the final code is #C987388A.



A.14 Bitwise Operations and Masking

Exercises found in Chapter 14 on page 154.

1. *Problem:* Perform the following bitwise operations on 8 bit numbers:

- (a) $5A_{16} \& 99_{16}$
- (b) $3B_{16} | 27_{16}$
- (c) $4F_{16} \oplus 31_{16}$
- (d) $\sim 5F_{16}$

Solution:

- (a) $5A_{16} \& 99_{16}$

First, convert both operands into binary. Combine each bit positionally using the AND operator (if both inputs are true, the result is true).

$$\begin{array}{r} 0101\ 1010 \\ \& 1001\ 1001 \\ \hline 0001\ 1000 \end{array}$$

Finally, convert back into the original base $0001\ 1000_2 = 18_{16}$.

- (b) $3B_{16} | 27_{16}$

First, convert both operands into binary. Combine each bit positionally using the OR operator (if either input is true, the result is true).

$$\begin{array}{r} 0011\ 1011 \\ | \quad 0010\ 0111 \\ \hline 0011\ 1111 \end{array}$$

Finally, convert back into the original base $0011\ 1111_2 = 3F_{16}$.

- (c) $4F_{16} \oplus 31_{16}$

First, convert both operands into binary. Combine each bit positionally using the XOR operator (exclusive-Or: if either but not both inputs are true, the result is true).

$$\begin{array}{r} 0100\ 1111 \\ \oplus \quad 0011\ 0001 \\ \hline 0111\ 1110 \end{array}$$

Finally, convert back into the original base $0111\ 1110_2 = 7E_{16}$.

- (d) $\sim 5F_{16}$

In order to complete the complement, we must know the number of bits. The problem told us eight bits, so we'll be sure to allocate the appropriate number. Unlike the other bitwise operators, the total number of bits must be known to arrive at the appropriate answer to the NOT operator.



$$\begin{array}{r} \sim 0101\ 1111 \\ \hline 1010\ 0000 \end{array}$$

Finally, convert back into the original base $1010\ 0000_2 = A0_{16}$.

An online webservice for retrieving aerial photography indicates what kind of images are available in a certain area using flags. The documentation states that following types exist:

Image Type	Flag Value (base 10)
B & W Photo	1
Topographic Map	2
Shaded Relief	4
Color Photo	8

2. *Problem:* For each of the following, find the appropriate flag value.

- (a) The area has shaded relief and color photo available.
- (b) The area has black and white photo, as well as color photo, available.
- (c) The area has all four types of imagery available.
- (d) No imagery is available for the area.

Solution: Starting with a decimal value of 0, add all the selected flag values.

- (a) The area has shaded relief and color photo available.
 $4 + 8 = 12$
- (b) The area has black and white photo, as well as color photo, available.
 $1 + 8 = 9$
- (c) The area has all four types of imagery available.
 $1 + 2 + 4 + 8 = 15$
- (d) No imagery is available for the area.
A value of 0, which has no bits turned on, indicates that no flags are selected.

Alternative Solution: Convert each flag value into its binary equivalent.

Image Type	Flag Value (Binary)
Black and White Photo	1
Topographic Map	10
Shaded Relief	100
Color Photo	1000

OR all selected flag values together.



- (a) The area has shaded relief and color photo available.

$$100 \mid 1000 = 1100_2 = 12_{10}$$

- (b) The area has black and white photo, as well as color photo, available.

$$1 \mid 1000 = 1001_2 = 9_{10}$$

- (c) The area has all four types of imagery available.

$$1 \mid 10 \mid 100 \mid 1000 = 1111_2 = 15_{10}$$

- (d) No imagery is available for the area.

As before, a value of 0 indicates no flags are selected.

3. *Problem:* Determine which types of imagery are available given the following values.

(a) 11_{10}

(b) 11_8

(c) F_{16}

(d) 0110_2

Solution: First, convert each value into binary. Each 1 bit indicates the presence of a flag value. Which flag value can be determined by comparing the position of the 1 bit to the list of flag values. More formally, it is possible to AND a flag value against the number to find out if the flag is present or not.

- (a) $11_{10} = 1011_2$. There are 1 bits in the 1, 2, and 8 position (binary place value), indicating that black and white, topographic, and color images are available. Shaded relief is not available; taking the binary value of shaded relief from the previous answer, we find that $1011_2 \& 100_2 = 0000_2$. A result of 0 indicates that flag is not set.
- (b) $11_8 = 1001_2$. There are 1 bits in the 1 and 8 position, indicating that black and white, as well as color, images are available.
- (c) $F_{16} = 1111_2$. All of the relevant bits are 1s, so all types of imagery are available.
- (d) 0110_2 . There are 1 bits in the 2 and 4 position, indicating that topographic map and shaded relief types of imagery are available.

4. *Problem:* How many devices can participate in an IPv4 network with a netmask of 255.255.255.240? (Excluding the use of any gateways, etc.)

Solution: In binary, the netmask is represented as 1111 1111.1111 1111.1111 1111.1111 0000₂. There are four zero bits, which indicate the bits that can change from one address to another and still be inside the same netmask. The first 28 bits must stay the same between two devices for those devices to directly communicate. Given that four bits are available to enumerate devices, there can be up to $2^4 = 16$ devices participating.



Alternative Solution: Find the bitwise NOT of the netmask. $\sim FFFFFFFF_{16} = 0000000F_{16} = 16_{10}$ different devices.

5. *Problem:* If a device has an IPv4 address of 10.5.77.203, and a netmask of 255.255.255.252, what is the range of IPv4 addresses it can directly communicate with?

Solution: The device IP address, in hexadecimal, is $0A054DCB_{16}$. The netmask is $FFFFFFFFFFC_{16}$. Thus, the lowest IP address in the range can be found by AND'ing the device IP with the netmask, and the highest address in the range can be found by OR'ing the device IP with the complement of the netmask.

Lowest: $0A054DCB_{16} \& FFFFFFFC_{16} = 0A054DC8_{16}$ (10.5.77.200)

Highest: $0A054DCB_{16} | \sim FFFFFFFC_{16} = 0A054DCB_{16} | 00000003_{16} = 0A054DCB_{16}$ (10.5.77.203)

Thus, there are four possible IP addresses in the range, 10.5.77.200 through 10.5.77.203.

6. *Problem:* A pixel of color #45BC09 is exclusive-OR'd with a background pixel of color #045A9F. What is the resulting color?

Solution: To find the new color, we must compute $45BC09_{16} \oplus 045A9F_{16}$. Convert both to binary.

$$\begin{array}{r} 100\ 0101\ 1011\ 1100\ 0000\ 1001 \\ \oplus\ 100\ 0101\ 1010\ 1001\ 1111 \\ \hline 100\ 0001\ 1110\ 0110\ 1001\ 0110 \end{array}$$

Converting back to hexadecimal, the resulting color is #41E696.

7. *Problem:* Using the XOR cipher, encrypt the UTF-8 phrase “Hello World” with the passcode “aBc”.

Solution: First, determine the code point and UTF-8 representation of each character (reference the chapter on Unicode). Each symbol in both the plaintext and the passcode can be represented with one byte (2 hexadecimal digits) in UTF-8. The plaintext is represented by $48656C6C6F\ 20\ 576F726C64_{16}$. The passcode is represented by 614263_{16} .

To apply the XOR cipher, we need to repeat the passcode to match the length of the plaintext.

$$\begin{array}{r} 48656C\ 6C6F20\ 576F72\ 6C64 \\ \oplus\ 614263\ 614263\ 614263\ 6142 \\ \hline 29270F\ 0D2D43\ 362D11\ 0D26 \end{array}$$



There is no reason to suspect that the resulting encrypted text is valid under any particular encoding, such as UTF-8. Thus, we leave it as a byte string.

Check: Decode the byte string $29270F\ 0D2D43\ 362D11\ 0D26_{16}$ with the passcode 614263_{16} .

$$\begin{array}{r} 29270F\ 0D2D43\ 362D11\ 0D26 \\ \oplus\ 614263\ 614263\ 614263\ 6142 \\ \hline 48656C\ 6C6F20\ 576F72\ 6C64 \end{array}$$

Confirm that the resulting byte string is valid UTF-8, and represents the phrase “Hello World”, which it does.

8. *Problem:* Using the XOR cipher, decrypt the content represented by $3A100E0F10091D_{16}$ using the UTF-8 passcode “xyz” and show the output using UTF-8.

Solution: The passcode “xyz” is represented in UTF-8 as $78797A_{16}$. Repeat the passcode until it aligns with the encrypted text.

$$\begin{array}{r} 3A100E\ 0F1009\ 1D \\ \oplus\ 78797A\ 78797A\ 78 \\ \hline 426974\ 776973\ 65 \end{array}$$

Refer to techniques for decoding UTF-8. This string decodes to “Bitwise”.



A.15 Error Correcting Codes

Exercises found in Chapter 15 on page 167.

1. *Problem:* Find the Hamming Distance between the two messages $0101\ 1001_2$ and $0111\ 1010_2$.

Solution: The Hamming Distance is the number of positions in which two messages vary. To find the Hamming distance, stack the messages and compare each bit column-wise to see if it is equal or not. Count all positions where the bits are not equal.

$$\begin{array}{cccccccc}
 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
 \hline
 = & = & \neq & = & = & = & \neq & \neq
 \end{array}$$

There are three positions where the messages differ, thus they have a Hamming Distance of three.

2. *Problem:* The following messages have an even parity bit appended to the end and were transmitted over a possibly noisy channel, with at most one error. Determine which messages contain an error.

- (a) $1111\ 1111_2$
- (b) $1010\ 1011_2$
- (c) $1111\ 0000_2$
- (d) $0000\ 0000_2$

Solution: Even parity means that with the parity bit added, the number of 1s in the encoded message should be an even number (0, 2, 4, ...). Count the total number of 1s in each message to determine if it is even (no error) or odd (error).

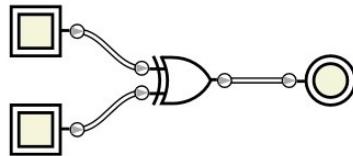
- (a) $1111\ 1111_2$
Eight ones is even; no error.
- (b) $1010\ 1011_2$
Five ones is odd; error.
- (c) $1111\ 0000_2$
Four ones is even; no error.
- (d) $0000\ 0000_2$
Zero ones is even; no error.

3. *Problem:* Construct a logic circuit which accepts 4 data bits as input and indicates the correct even parity bit for these four data bits.

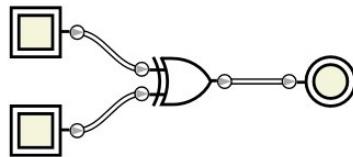
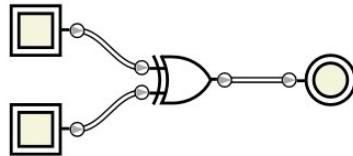
Solution: The book notes that parity can be computed using XOR gates. Recall the definition of exclusive-OR is that it is true if one or the other, but not both, inputs are true. Thus,



a 2 bit input set could be put into a single XOR gate to directly compute the even parity of those two bits (because if both bits are 0, or both bits are 1, the output will be 0 and the total number of 1s will be even; on the other hand, if one input is 1 and one input is 0, the output is 1, bringing the total number of 1s to two, which is even).



This operation could be duplicated to bring the total to four inputs (as desired). However, how can the two outputs be finally combined?

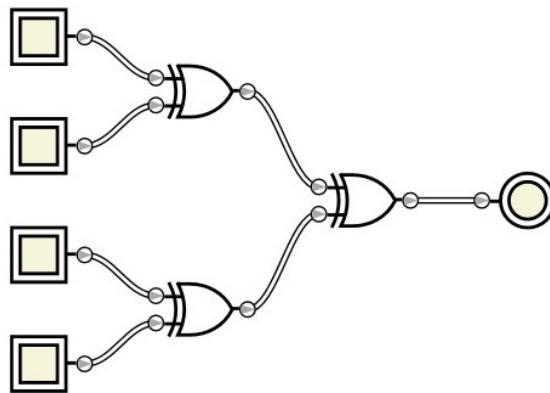


The output from the XOR gates can also be thought of as an indicator: false (0) means even, true (1) means odd. If we have two blocks of two inputs each, then there are four ways they could be combined:

$$\begin{array}{l}
 \text{Even} + \text{Even} = \text{Even} \\
 \text{Even} + \text{Odd} = \text{Odd} \\
 \text{Odd} + \text{Even} = \text{Odd} \\
 \text{Odd} + \text{Odd} = \text{Even}
 \end{array}$$

This result, using truth values for even and odd, is itself nothing more than exclusive-OR. Therefore, the individual two-bit outputs can be finally combined with an XOR to calculate the even parity bit.





4. *Problem:* The following messages, each with 8 data bits, have been encoded using the Berger code and transmitted over a channel which may only convert 1s to 0s. Determine which messages contain error(s).

- (a) 0101 1001 0100₂
- (b) 1101 0011 0001₂
- (c) 0011 0111 0010₂
- (d) 1110 0111 0010₂

Solution: Each message can be divided into data and error detection bits. The first eight bits are data; the last four are error correction (check bits). To determine if a message has errors, add up the number of 0s in the data and compare this value as an unsigned integer to the remaining four bits.

- (a) 0101 1001 | 0100₂
Four zeros found in data, so check is expected to be $4_{10} = 0100_2$. This matches, so no error exists.
- (b) 1101 0011 | 0001₂
Three zeros found in data, so check is expected to be $3_{10} = 0011_2$. However, the check does not match, so one or more errors exist.
- (c) 0011 0111 | 0010₂
Three zeros found in data, so check is expected to be $3_{10} = 0011_2$. However, the check does not match, so one or more errors exist.
- (d) 1110 0111 | 0010₂
Two zeros found in data, so check is expected to be $2_{10} = 0010_2$. This matches, so no error exists.

5. *Problem:* Encode the following messages using the H(7,4) Hamming code.

- (a) 0001₂



- (b) 0000_2
- (c) 1110_2
- (d) 1010_2

Solution: For each block of data, the three parity bits will be calculated and then the final message constructed. Recall the parity bits are calculated $p_1 = d_1 \oplus d_2 \oplus d_4$, $p_2 = d_1 \oplus d_3 \oplus d_4$, and $p_3 = d_2 \oplus d_3 \oplus d_4$. The message is then laid out with the pattern $p_1 p_2 d_1 p_3 d_2 d_3 d_4$.

- (a) 0001_2
Parity bits $p_1 = 0 \oplus 0 \oplus 1 = 1$, $p_2 = 0 \oplus 0 \oplus 1 = 1$, $p_3 = 0 \oplus 0 \oplus 1 = 1$. The encoded message is thus 1101001_2 .
- (b) 0000_2
Parity bits $p_1 = 0 \oplus 0 \oplus 0 = 0$, $p_2 = 0 \oplus 0 \oplus 0 = 0$, $p_3 = 0 \oplus 0 \oplus 0 = 0$. The encoded message is thus 0000000_2 .
- (c) 1110_2
Parity bits $p_1 = 1 \oplus 1 \oplus 0 = 0$, $p_2 = 1 \oplus 1 \oplus 0 = 0$, $p_3 = 1 \oplus 1 \oplus 0 = 0$. The encoded message is thus 0010110_2 .
- (d) 1010_2
Parity bits $p_1 = 1 \oplus 0 \oplus 0 = 1$, $p_2 = 1 \oplus 1 \oplus 0 = 0$, $p_3 = 0 \oplus 1 \oplus 0 = 1$. The encoded message is thus 1011010_2 .

6. *Problem:* The following messages have been encoded with the H(7,4) Hamming code and transmitted over a possibly noisy channel, with at most one error. Determine which messages contain an error. In all cases, find the original 4-bit message.

- (a) 1010101_2
- (b) 1111111_2
- (c) 1110010_2
- (d) 0110100_2

Solution: To detect and correct an error, the data and parity bits must first be separated. Then, the expected parity bits will be computed and compared to the received parity bits.

- (a) 1010101_2
The data bits are 1101_2 , with parity $r_1 = 1, r_2 = 0, r_3 = 0$. To detect an error, compute parity $p_1 = 1 \oplus 1 \oplus 1 = 1$, $p_2 = 1 \oplus 0 \oplus 1 = 0$, and $p_3 = 1 \oplus 0 \oplus 1 = 0$. Confirm that $r_1 = p_1$, $r_2 = p_2$, and $r_3 = p_3$. Therefore, there is no error.
- (b) 1111111_2
The data bits are 1111_2 , with parity $r_1 = 1, r_2 = 1, r_3 = 1$. Note that each parity bit is calculated based on the exclusive-OR of three data bits. Since all data bits are 1, the only possible parity setup is $1 \oplus 1 \oplus 1 = 1$. Therefore, all parity bits must be 1 to indicate no error. This is exactly the case seen, therefore, there is no error.



(c) 1110010_2

The data bits are 1010_2 , with parity $r_1 = 1, r_2 = 1, r_3 = 0$. Compute parity $p_1 = 1 \oplus 0 \oplus 0 = 1, p_2 = 1 \oplus 1 \oplus 0 = 0$, and $p_3 = 0 \oplus 1 \oplus 0 = 1$. Compare calculated parity to received parity, and note that $r_1 = p_1$, but $r_2 \neq p_2$ and $r_3 \neq p_3$. The bit position of the error is found by adding the positions of the non-matching parity bits, with 1 being the leftmost position: parity bit 2 is in position 2, and parity bit 3 is in position 4. The error is in bit $2 + 4 = 6$ of the original message. To correct the error, flip this bit: $11100\textcolor{orange}{0}0_2$. The corrected data bits are 1000_2 .

(d) 0110100_2

The data bits are 1100_2 , with parity $r_1 = 0, r_2 = 1, r_3 = 0$. Compute parity $p_1 = 1 \oplus 1 \oplus 0 = 0, p_2 = 1 \oplus 0 \oplus 0 = 1$, and $p_3 = 1 \oplus 0 \oplus 0 = 1$. Compare calculated parity to received parity, and note that $r_1 = p_1$ and $r_2 = p_2$, but $r_3 \neq p_3$. The location of the error is indicated by a sum of the positions of the erroneous parity bits. When only one parity bit indicates an error, the position will be that parity bit itself. Thus, the error in this message is actually in the third parity bit. To correct the error, flip the bit: $011\textcolor{orange}{1}100_2$. However, this error does not change the data bits.

7. *Problem:* Construct the array for the Hadamard code RM(16,5).

Solution: Each Hadamard array is based on three copies (one inverted, in the lower right) of the previous array. The previous array, RM(8,4), is shown in the text.

00000	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
00001	1 0 1 0 1 0 1 0	1 0 1 0 1 0 1 0
00010	1 1 0 0 1 1 0 0	1 1 0 0 1 1 0 0
00011	1 0 0 1 1 0 0 1	1 0 0 1 1 0 0 1
00100	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0
00101	1 0 1 0 0 1 0 1	1 0 1 0 0 1 0 1
00110	1 1 0 0 0 0 1 1	1 1 0 0 0 0 1 1
00111	1 0 0 1 0 1 1 0	1 0 0 1 0 1 1 0
01000	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
01001	1 0 1 0 1 0 1 0	0 1 0 1 0 1 0 1
01010	1 1 0 0 1 1 0 0	0 0 1 1 1 0 0 1
01011	1 0 0 1 1 0 0 1	0 1 1 0 0 1 1 0
01100	1 1 1 1 0 0 0 0	0 0 0 0 0 1 1 1
01101	1 0 1 0 0 1 0 1	0 1 0 1 1 0 1 0
01110	1 1 0 0 0 0 1 1	0 0 1 1 1 1 0 0
01111	1 0 0 1 0 1 1 0	0 1 1 0 1 0 0 1
10000	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
10001	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1
10010	0 0 1 1 0 0 1 1	0 0 1 1 1 0 0 1
10011	0 1 1 0 0 1 1 0	0 1 1 0 0 1 1 0
10100	0 0 0 0 1 1 1 1	0 0 0 0 0 1 1 1
10101	0 1 0 1 1 0 1 0	0 1 0 1 1 0 1 0
10110	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0
10111	0 1 1 0 0 0 1 0	0 1 1 0 1 0 0 1
11000	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1
11001	0 1 0 1 0 1 0 1	1 0 1 0 1 0 1 0
11010	0 0 1 1 0 0 1 1	1 1 0 0 1 1 0 0
11011	0 1 1 0 0 1 1 0	1 0 0 1 1 0 0 1
11100	0 0 0 0 1 1 1 1	1 1 1 1 1 0 0 0
11101	0 1 0 1 1 0 1 0	1 0 1 0 0 0 1 0
11110	0 0 1 1 1 1 0 0	1 1 0 0 0 0 1 1
11111	0 1 1 0 1 0 0 1	1 0 0 1 0 1 1 0



8. *Problem:* The following messages have been encoded using the RM(16,5) Hadamard code and transmitted over a possibly noisy channel, with up to four errors. Determine which messages contain error(s) and how many error(s) are present. In cases with up to three errors, find the original 4-bit message. In cases with four errors, show why recovering the original 5-bit message is not possible.
- (a) 1010 1010 1010 1010₂
 - (b) 0111 1000 0000 0111₂
 - (c) 0011 1110 1100 0011₂
 - (d) 0011 1100 0011 0000₂
 - (e) 1000 1110 1101 0100₂

Solution: Find the Hamming Distance (number of positions where bits differ) between the received message and each row in the RM(16,5) array. The row with the minimal Hamming Distance (minimum number of positions where bits differ) indicates the corrected data bits. In these solutions, highlighted bits are those that differ from the received encoded message.



(a) 1010 1010 1010 1010₂

Find Hamming Distance for each RM(16,5) row.

00000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	8
00001	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0
00010	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	8
00011	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	8
00100	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	8
00101	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	8
00110	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	8
00111	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	8
01000	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	8
01001	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	8
01010	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	1	8
01011	1	0	0	1	1	0	0	0	1	0	1	1	0	0	1	1	8
01100	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	8
01101	1	0	1	0	0	1	0	1	0	1	0	1	1	0	1	0	8
01110	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0	8
01111	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	8
10000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8
10001	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	16
10010	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	8
10011	0	1	1	0	0	1	1	1	0	0	1	1	1	0	1	1	8
10100	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	8
10101	0	1	0	1	1	0	1	0	0	0	1	0	1	1	0	1	8
10110	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	0	8
10111	0	1	1	0	1	0	0	0	1	0	1	1	1	0	0	1	8
11000	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	8
11001	0	1	0	1	0	1	0	1	1	1	0	1	1	0	1	0	8
11010	0	0	1	1	0	0	1	1	1	1	0	0	1	1	1	0	8
11011	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	8
11100	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	8
11101	0	1	0	1	1	0	1	0	1	0	1	0	0	1	0	1	8
11110	0	0	1	1	1	1	0	0	1	1	0	0	0	0	1	1	8
11111	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	8

The row with the lowest Hamming distance corresponds to the correct message. The Hamming distance itself indicates the number of errors. In this case, the message 00001₂ was received without error.



(b) 0111 1000 0000 0111₂

Find Hamming Distance for each RM(16,5) row.

00000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	9
00001	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	9
00010	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	9
00011	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	9
00100	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	9
00101	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	9
00110	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	1	9
00111	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	9
01000	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	7
01001	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	7
01010	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	1	7
01011	1	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	7
01100	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	3	
01101	1	0	1	0	0	1	0	1	0	1	0	1	1	0	1	0	11
01110	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0	11
01111	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	11
10000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7
10001	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	7
10010	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	1	7
10011	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	7
10100	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	7
10101	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0	7
10110	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	7
10111	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	7
11000	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	9
11001	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	9
11010	0	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	9
11011	0	1	1	0	0	1	1	1	0	1	0	0	1	1	0	1	9
11100	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	13
11101	0	1	0	1	1	0	1	0	1	0	1	0	0	0	1	0	5
11110	0	0	1	1	1	1	1	0	0	1	1	0	0	0	0	1	5
11111	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	5

The row with the lowest Hamming distance corresponds to the correct message. The Hamming distance itself indicates the number of errors. In this case, the message 01100₂ was received with three errors (the maximum correctable by this particular code).

(c) 0011111011000011₂

Find Hamming Distance for each RM(16,5) row.

00000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7
00001	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	7
00010	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	9
00011	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	9
00100	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	9
00101	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	9
00110	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1	7
00111	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	7
01000	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	7
01001	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	7
01010	1	1	0	0	1	1	0	0	0	0	1	1	0	0	0	1	9
01011	1	0	0	1	1	0	0	0	1	0	1	1	0	0	1	1	9
01100	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	9
01101	1	0	1	0	0	1	0	1	0	1	0	1	1	0	1	0	9
01110	1	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	15
01111	1	0	0	1	0	1	1	0	0	0	1	1	0	1	0	1	7
10000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
10001	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	9
10010	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	7
10011	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	7
10100	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	7
10101	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0	7
10110	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	9
10111	0	1	1	0	1	0	0	1	0	1	1	1	0	1	0	1	9
11000	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	9
11001	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	9
11010	0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0	7
11011	0	1	1	0	0	1	1	1	0	1	0	0	1	1	0	1	7
11100	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	7
11101	0	1	0	1	1	0	1	0	1	0	1	0	0	1	0	1	7
11110	0	0	1	1	1	1	1	0	0	1	1	1	0	0	0	1	1
11111	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	9

In this case, the message 11110₂ was received with one error.

(d) 0011110000110000_2

Find Hamming Distance for each RM(16,5) row.

00000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	10
00001	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	8
00010	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	10
00011	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	8
00100	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	6
00101	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	8
00110	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	14
00111	1	0	0	1	0	1	1	0	1	0	0	1	0	1	0	8
01000	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	6
01001	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	8
01010	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	6
01011	1	0	0	1	1	0	0	1	0	1	1	0	0	1	0	8
01100	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	10
01101	1	0	1	0	0	1	0	1	0	1	0	1	1	0	1	8
01110	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	10
01111	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	8
10000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
10001	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	8
10010	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	6
10011	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	8
10100	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	10
10101	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	8
10110	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	2
10111	0	1	1	0	1	0	0	1	0	1	1	0	1	0	1	8
11000	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	10
11001	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	8
11010	0	0	1	1	0	0	1	1	1	1	1	0	0	1	1	10
11011	0	1	1	0	0	1	1	0	1	0	0	1	1	0	1	8
11100	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	6
11101	0	1	0	1	1	0	1	0	1	0	1	0	0	1	0	8
11110	0	0	1	1	1	1	0	0	1	1	0	0	0	1	1	6
11111	0	1	1	0	1	0	0	1	1	0	0	1	0	1	0	8

In this case, the message 10110_2 was received with two errors.

(e) $1000\ 1110\ 1101\ 0100_2$

Find Hamming Distance for each RM(16,5) row.

00000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	8
00001	1	0	1	0	1	1	0	1	0	1	0	1	0	1	1	0	8
00010	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	0	4
00011	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	0	8
00100	1	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0	8
00101	1	0	1	0	0	1	0	1	1	0	0	1	0	0	1	0	8
00110	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	8
00111	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	4
01000	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	8
01001	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	4
01010	1	1	0	0	1	1	1	0	0	0	0	1	1	0	0	1	8
01011	1	0	0	1	1	0	0	0	1	0	1	1	0	0	1	1	8
01100	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	12
01101	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	8
01110	1	1	0	0	0	0	1	1	0	0	0	1	1	1	1	0	8
01111	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	8
10000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8
10001	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	8
10010	0	0	1	1	0	0	1	1	0	0	1	1	1	0	0	1	12
10011	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	8
10100	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	8
10101	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0	8
10110	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1	0	8
10111	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	12
11000	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	8
11001	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	12
11010	0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0	8
11011	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	8
11100	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	4
11101	0	1	0	1	1	0	1	0	1	0	0	1	0	0	1	0	8
11110	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	8
11111	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	8

The row with the lowest Hamming distance corresponds to the correct message. The Hamming distance itself indicates the number of errors. In this case, however, there is no single row with the lowest Hamming distance. Instead, three rows, 00010_2 , 00111_2 , 01001_2 , and 11100_2 all share the same Hamming distance of four. With no additional information, we are unable to determine the correct original message from this group of candidates, and so four errors cannot be corrected.

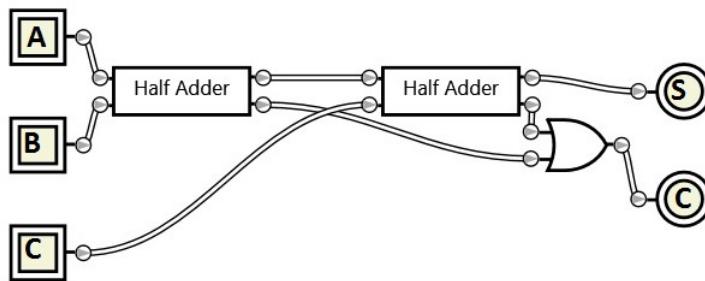


A.16 Digital Logic

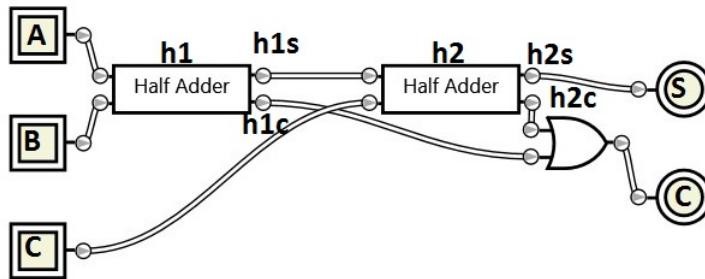
Exercises found in Chapter 17 on page 195.

1. *Problem:* Using the logical expressions for the half adder ($s = a \oplus b$ and $c = a \wedge b$), prove the correctness of the construction of the full adder based on two half adders and an OR gate.

Solution: The diagram shows how two half adders and an OR gate were combined to form a full adder:



We can define the outputs s and c with respect to the half adders. Let h_1 and h_2 represent the two half adders. The respective sum and carry outputs from can further be represented as h_{1s} , h_{1c} , h_{2s} , h_{2c} .



Following the circuit diagram, we can define $h_{1s} = a \oplus b$, $h_{1c} = a \wedge b$, $h_{2s} = h_{1s} \oplus c$, and $h_{2c} = h_{1s} \wedge c$ respectively. The final results can be defined as $s_{out} = h_{2s}$ and $c_{out} = h_{1c} \vee h_{2c}$.

Expanding the definition of s_{out} and c_{out} by substitution yields $s_{out} = h_{1s} \oplus c = a \oplus b \oplus c$ and $c_{out} = h_{1c} \vee h_{2c} = (a \wedge b) \vee (h_{1s} \wedge c) = (a \wedge b) \vee ((a \oplus b) \wedge c)$.

Recall the table for three bit addition:

0	+	0	+	0	=	0
0	+	1	+	0	=	1
0	+	0	+	1	=	1
0	+	1	+	1	=	10
1	+	0	+	0	=	1
1	+	1	+	0	=	10
1	+	0	+	1	=	10
1	+	1	+	1	=	11



Rewrite this sum series as a truth table.

a	b	c	Carry	Sum
F	F	F	F	F
F	T	F	F	T
F	F	T	F	T
F	T	T	T	F
T	F	F	F	T
T	T	F	T	F
T	F	T	T	F
T	T	T	T	T

To prove that the given circuit implements this truth table, determine the values of s_{out} and c_{out} and show that they match Sum and Carry. The value of s_{out} is true whenever an odd number of inputs are true (given the nature of XOR).

a	b	c	Carry	Sum	c_{out}	s_{out}
F	F	F	F	F		F
F	T	F	F	T		T
F	F	T	F	T		T
F	T	T	T	F		F
T	F	F	F	T		T
T	T	F	T	F		F
T	F	T	T	F		F
T	T	T	T	T		T

The Sum output matches s_{out} . Consider that c_{out} is true whenever both a and b are true; and also when c is true and at least one of a or b is true.

a	b	c	Carry	Sum	c_{out}	s_{out}
F	F	F	F	F	F	F
F	T	F	F	T	F	T
F	F	T	F	T	F	T
F	T	T	T	F	T	F
T	F	F	F	T	F	T
T	T	F	T	F	T	F
T	F	T	T	F	T	F
T	T	T	T	T	T	T

The Carry out matches c_{out} . Thus, the construction of the full adder is proven correct.

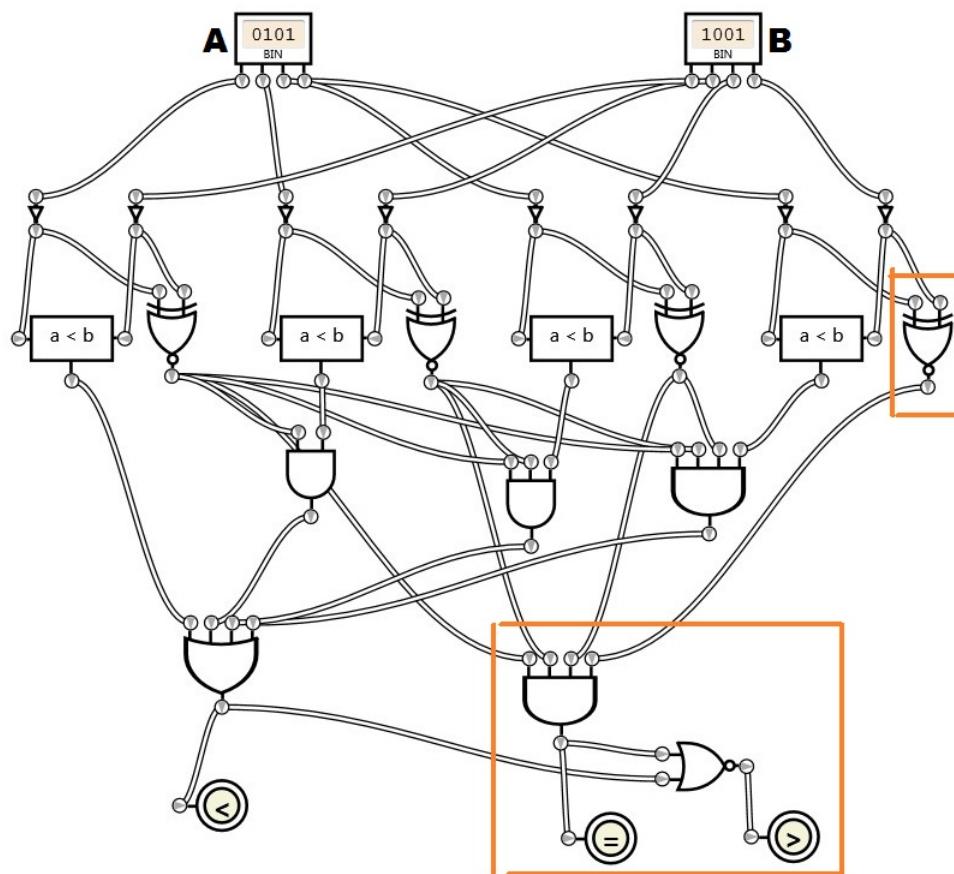


2. *Problem:* Extend the four bit comparison circuit shown in the chapter to have three outputs: one for $a < b$ (existing), one for $a = b$, and one for $a > b$. As much as possible, re-use the existing circuit.

Solution: The given circuit shows one output, which is true when $a < b$. How can $a = b$ be determined? Note previously that XNOR was used for equality. Investigating the existing circuit, we can see that several XNOR gates using appropriate input pairs already exist. We need only to add an additional XNOR for the rightmost bit, and then combine all four XNORs together with an AND. If all four XNORs are true, the inputs are equal.

Now that we have $a < b$ and $a = b$, how can $a > b$ be found? No real additional work is necessary: simply note that $a > b$ is true if and only if neither $a < b$ nor $a = b$.

The revised circuit is shown. A box indicates the new gates, which include only one AND, one NOR, and one XNOR.



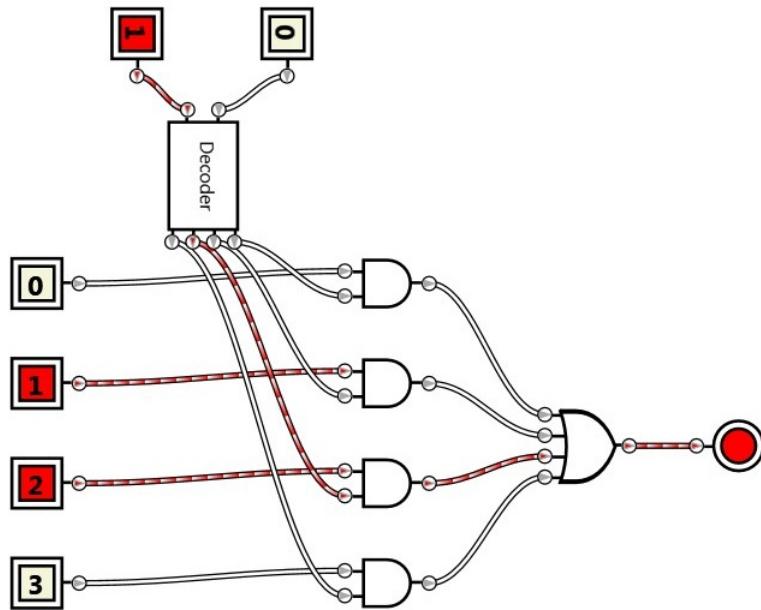
3. *Problem:* A multiplexer is a logic circuit that has n select inputs and 2^n data inputs. The data input corresponding to the given select value is sent to the single output line. Implement a 4 to 1 multiplexer; that is, a multiplexer with 4 data inputs and one output.



Solution: Using the 2 to 4 decoder, we must set $n = 2$, as the decoder has 2 select inputs and 4 individual “values”. For each output of the decoder, we can AND it with the corresponding input to the multiplexer, and then OR all the results together.

This will work because only one (and exactly one) output from the decoder will be true. The rest will be false. The false outputs will also result in a false out of an AND gate; the true output will then pass through that particular input value. The OR gate will, at most, have one true input.

Using the decoder as a module, a 4 to 1 multiplexer can be constructed around it. In this case, the multiplexer is shown with input $10_2 = 2_{10}$ selected. Input 2 is true, so the output is true.



4. *Problem:* A clocked T Flip-Flop (T stands for Toggle) has two inputs: T and Clock. It will invert its output if the T is true on the clock edge.

- (a) Implement a T Flip-Flop.
- (b) If the T input is held true, the output of the T Flip-Flop should alternate at half the rate of the clock input. Confirm this behavior in your implementation.

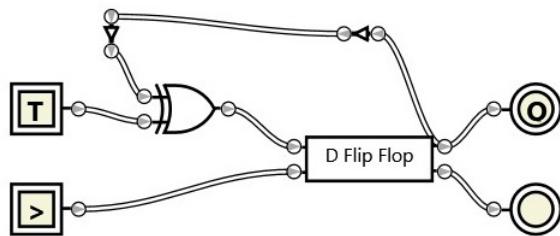
Solution: A T Flip-Flop should alter its state based on the request of the toggle input and of the current data. In order to discover the implementation of a T Flip-Flop, first consider what behavior it should have. If the toggle request is false, then the new value should match the current value (no change). If the toggle request is true, the new value should be the opposite (toggle) of the current value.



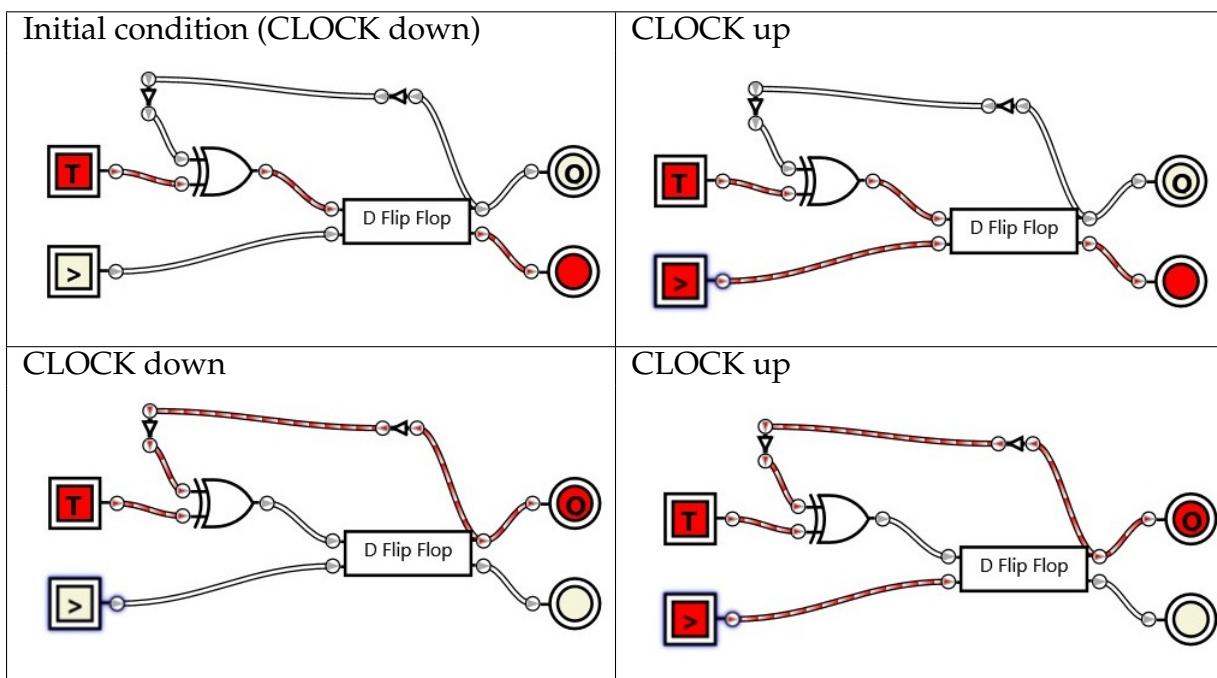
Toggle	Current Value	New Value
T	T	F
T	F	T
F	T	T
F	F	F

What operator does this table reflect? It is equivalent to the XOR operator.

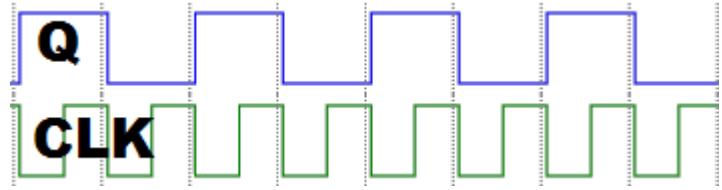
We can then build the T Flip-Flop based on an existing D Flip-Flop. The Data input D will be the XOR of the toggle input and the current Q data output.



To verify the operation of halving the clock speed, observe the behavior of the Q output when the toggle input is held true and the clock is alternated.

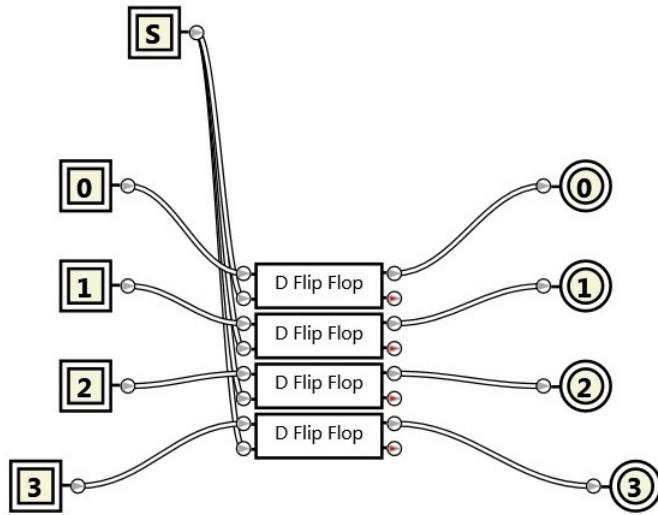


Another approach to verify the behavior is to graph the values of the clock (bottom) and the Q output (top) of the T Flip-Flop. Notice that the output alternates at half the rate of the clock input.



5. *Problem:* Implement an addressable memory bank. Provide for four storage locations (addressable by two bits), with four bits of data held in each storage location. When an address is selected, the current value in that location should be provided. It should also be possible to set a value into the addressed location.

Solution: A significant sequence of steps will be required to implement this specification. First, a four bit memory cell must be devised. The four bit memory cell can be implemented using 4 D Flip-Flops. The data inputs and outputs will go to four data inputs and four data outputs, and all the clock inputs will be synchronized to a single "Set" input.



The basic construct of the memory system involved four of these four bit memory cells.

To retrieve a value from memory, we will also need four multiplexers, one for each bit in a memory cell. Each multiplexer will need four inputs, one for each memory cell. Thus the multiplexers will be used to select which memory cell to retrieve from. These will then be connected to the output.

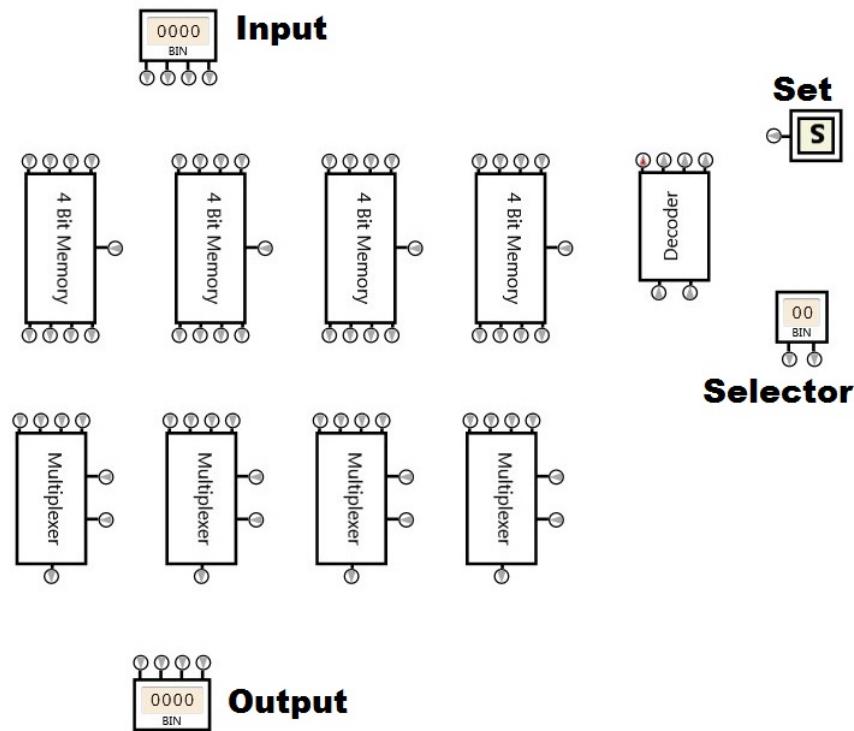
A selector (2 bit) will be needed to indicate which of the four memory cells to read/write from/to. The selector will connect to the multiplexers to indicate which memory cell is in use.

In order to set a value into memory, a set command (which will serve as the clock), and a decoder which will route the set command to the right 4 bit memory cell, will be needed.

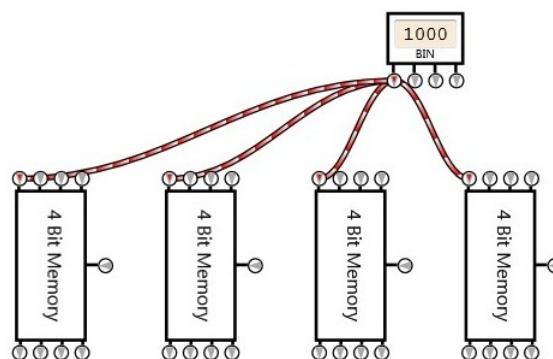


Additionally, a 4 bit data input block will be required. The decoder will use the same selector as the output to route the value to a memory cell.

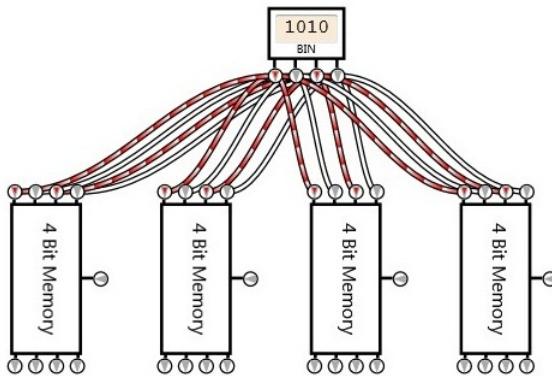
The pre-connection mockup of the 16 bit (4 by 4) memory grid can be prepared:



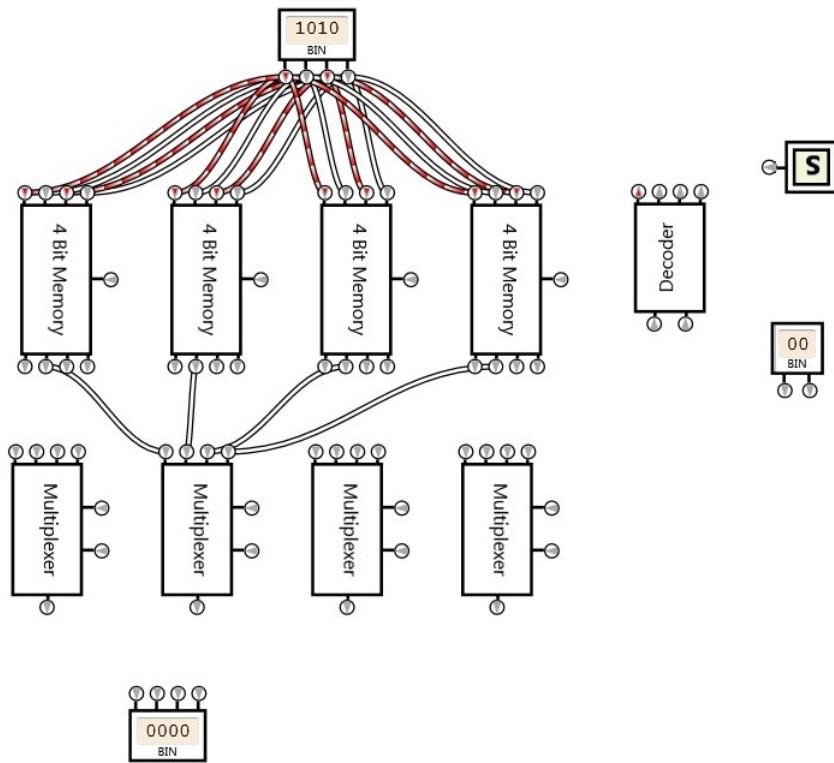
First, we will connect the data input to each respective position on the memory cells. The leftmost bit of input will connect to the leftmost input bit on each of the 4 bit memory cells. We see this first with just the leftmost bit connected:



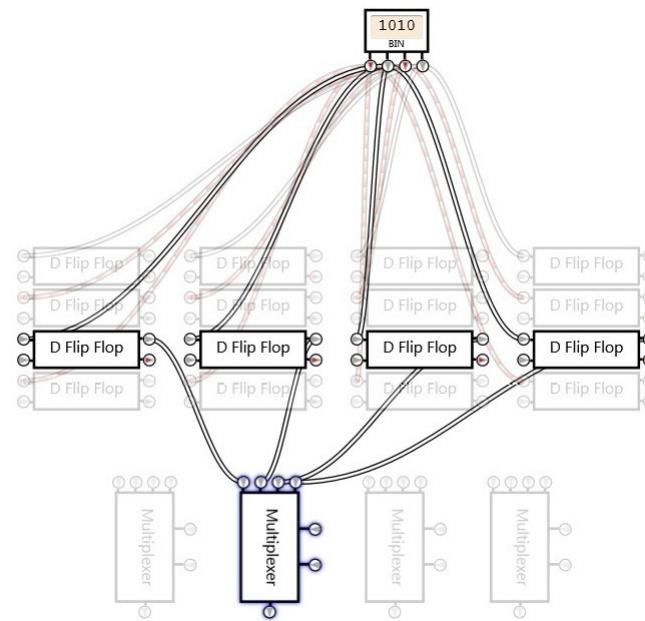
Then with all input bits connected to their respective targets.



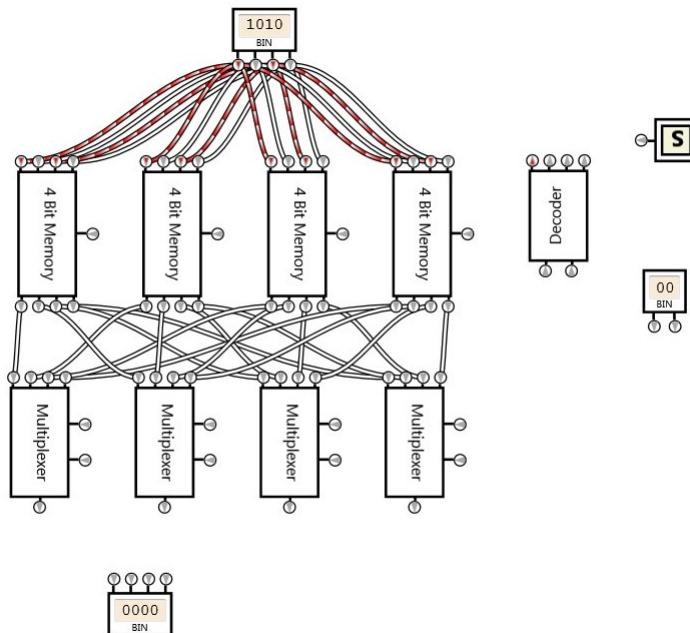
Next, the memory cell outputs can be connected to the multiplexers. Each memory cell will connect to each multiplexer; the left most bit on each memory cell will connect, in sequence, to the leftmost multiplexer, and so on. The second bit from the left is shown here connected to the second multiplexer.



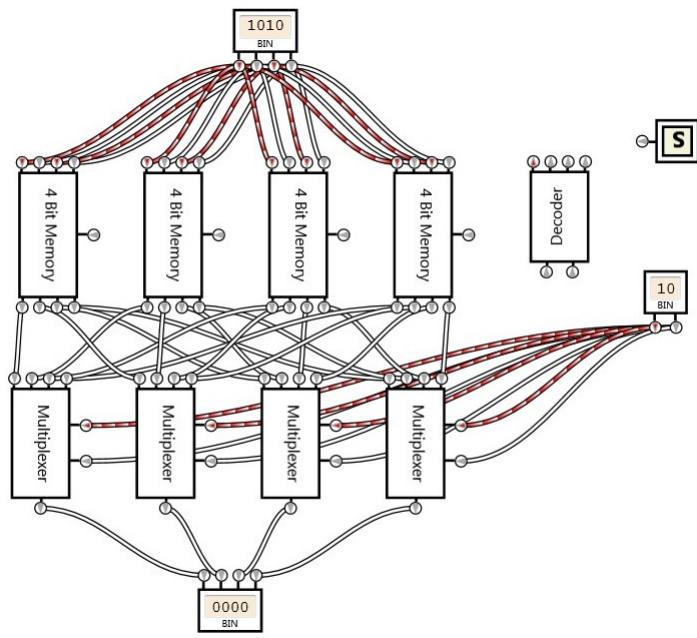
To see the implication of this connection, recall that each 4 bit memory cell is composed of four D Flip-Flops. The multiplexer cuts across a “row” of flip flops to extract the overall value of that bit. Here is the same circuit, with the memory cells expanded to be their representative flip flops, and the specific multiplexer highlighted, with the flip flops it references.



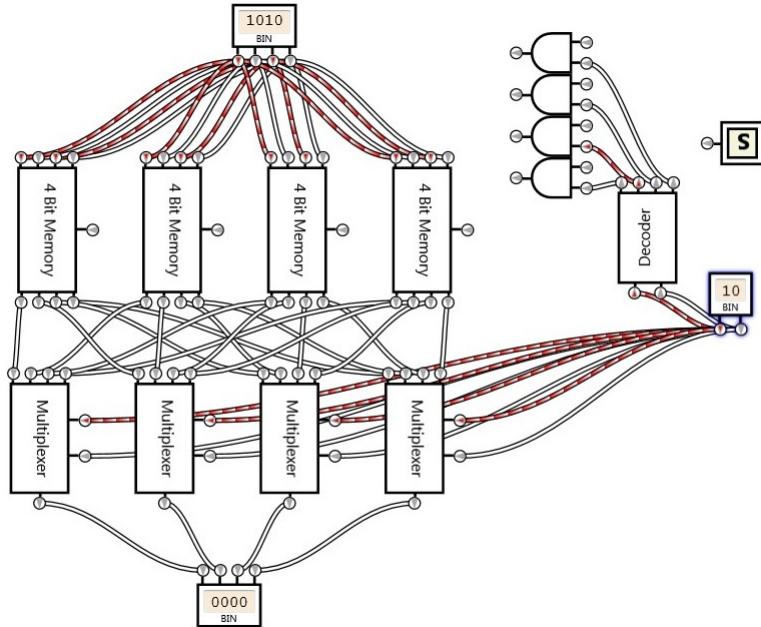
We will connect the remaining multiplexers likewise. The pattern appears to be a cross weave due to the nature of the connections.



Next, connect the selector to each multiplexer. This tells each multiplexer which memory cell to select from. Recall that each multiplexer is connected to each memory cell, so they all need to know which of those memory cells actually contains the value of current interest.



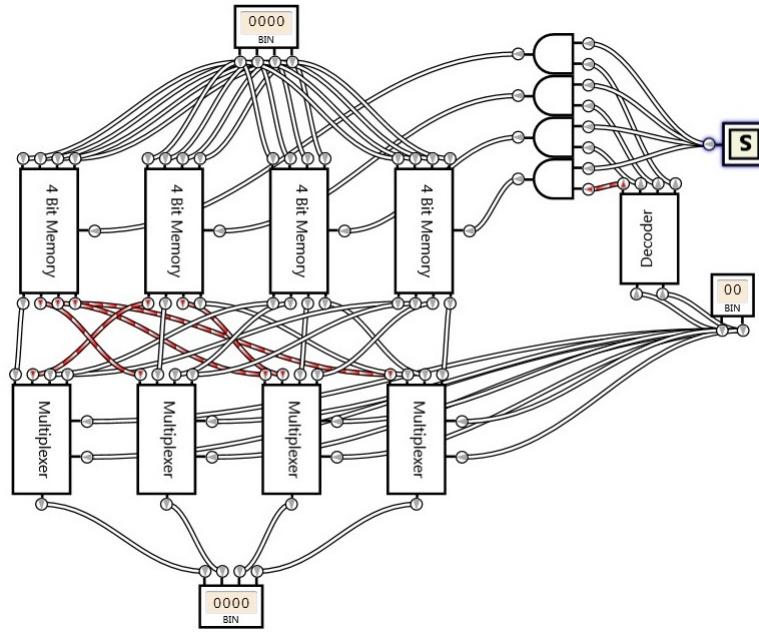
To set a value into memory, the decoder can be prepared to cause the "Set" input on one of the four bit memory cells to become true. The AND gates are added so that simply entering the value into the selector is not sufficient (otherwise memory would always update whenever a different cell was selected).



In order to perform an order, the Set command must be given. The set command will be wired to each AND gate to enable the signal to pass on to the clock on the respective 4 bit



memory cell. Finally, the AND gates will connect to the “Set” input of each 4 bit memory cell.



To enter a value into memory:

- Enter the 2 bit address into the selector.
- Enter the new 4 bit value into the input at the top.
- Press the Set command (S).
- Release the Set command.
- The output will now show the specified value whenever the selector indicates that cell.

To read a value from memory, simply update the selector, and then appropriate value will appear in the output.

Consider the four bit add/subtract circuit shown earlier in the chapter.

6. *Problem:* Show how the circuit processes the following inputs. Is the result correct or not? If not, why not?
 - $0101_2 + 0011_2$
 - $0101_{2C} + 0100_{2C}$

(c) $0101_{2C} - 0001_{2C}$

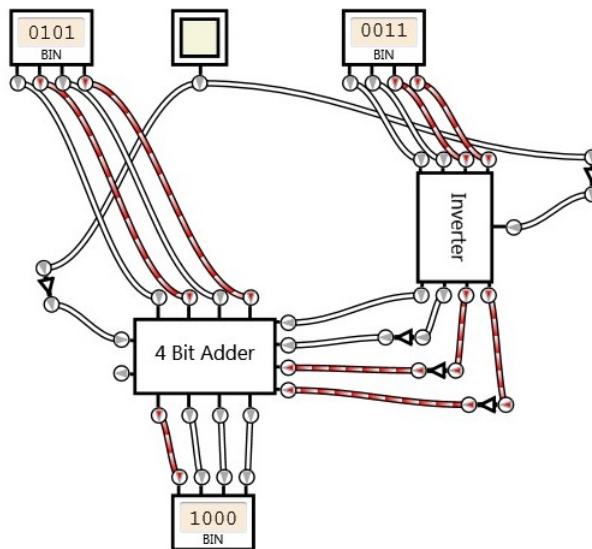
(d) $1110_2 + 1100_2$

(e) $1110_{2C} + 1100_{2C}$

(f) $1010_{2C} - 1100_{2C}$

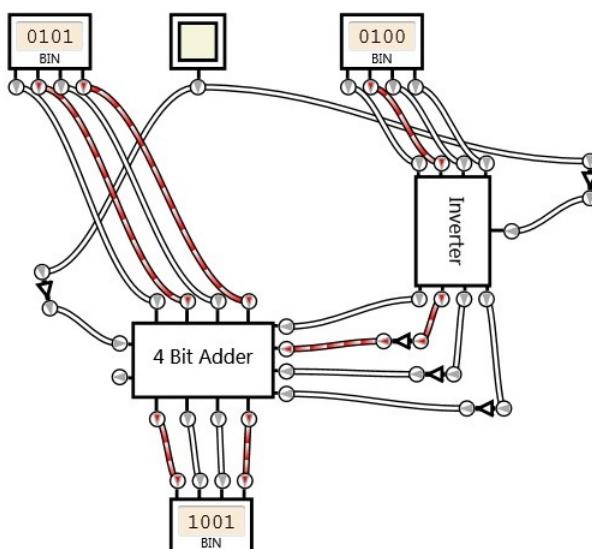
Solution: For each value, examine the circuit and its result.

(a) $0101_2 + 0011_2$



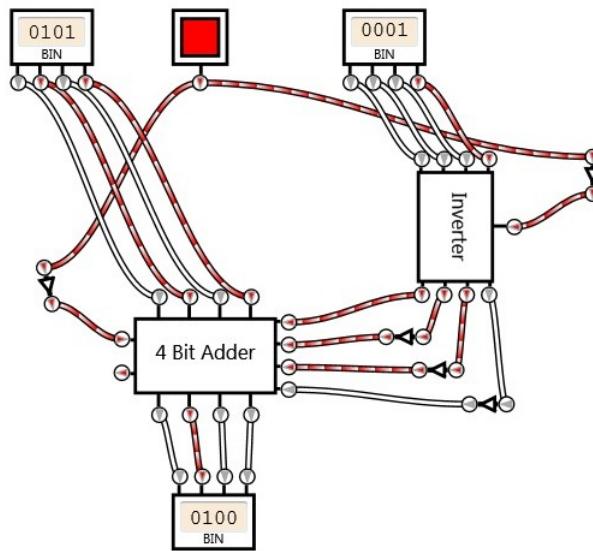
The adder correctly determines that $0101_2 + 0011_2 = 1000_2$.

(b) $0101_{2C} + 0100_{2C}$



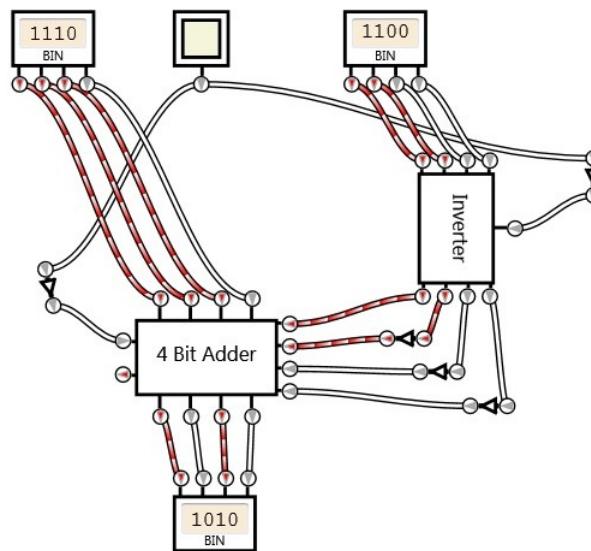
The adder claims that $0101_{2C} + 0100_{2C} = 1001_{2C}$. However, this is not correct. In decimal, the claim is that $5 + 4 = -7$. We can see that this claim is false because both inputs are positive, but the result is negative. The reason for this fault is that four bit Two's Complement cannot represent numbers larger than 7. The correct answer would be 9, but it exceeds the maximum positive value and so the value rolls over into negatives.

- (c) $0101_{2C} - 0001_{2C}$



The adder successfully calculates that $0101_{2C} - 0001_{2C} = 0100_{2C}$.

- (d) $1110_2 + 1100_2$



The adder claims that $1110_2 + 1100_2 = 1010_2$. However, this is not correct. In decimal, the claim is $14 + 12 = 10$. The addition fails because unsigned four bit numbers

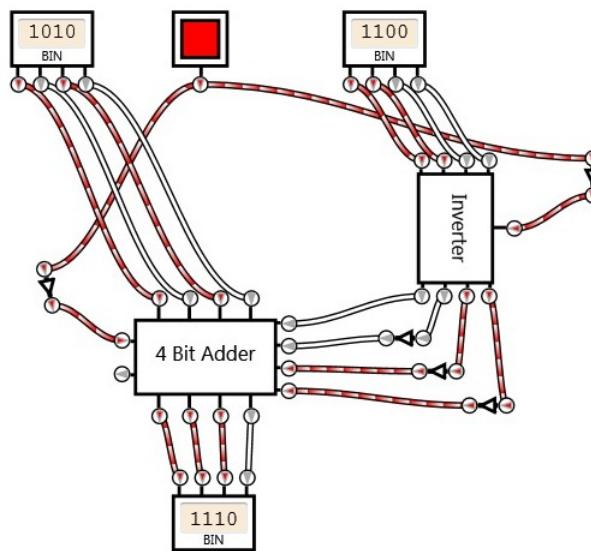
can only represent the range zero through fifteen. The actual result, 26, is not representable.

(e) $1110_{2C} + 1100_{2C}$

Notice that the actual function of the adder is identical to the previous statement. In this case, we simply interpret the inputs and results differently. The claim that $1110_{2C} + 1100_{2C} = 1010_{2C}$ is now valid and correct. Now the decimal claim is that $-2 + (-4) = -6$.

In general, an operation that may be correct in unsigned values may be incorrect in signed values, and vice versa.

(f) $1010_{2C} - 1100_{2C}$



Here the adder claims that $1010_{2C} - 1100_{2C} = 1110_{2C}$. In decimal, this claim is $-6 - (-4) = -2$. Although these two values could not be added together successfully (as the result would be -10, exceeding the representation range of four bit Two's Complement), they can be subtracted, with a correct result.

7. *Problem:* Devise an improvement to this circuit that will detect if an addition or subtraction result is incorrect due to insufficient bits to represent the result. Assume all values are Two's Complement.

Solution: As indicated in the text, the overflow result is not meaningful when dealing with Two's Complement. Another technique for detecting errors must be devised.

First, consider the possible correct output numbers based on the inputs. Note that if the two inputs have the same sign, the output should match that sign. That is, if two positives are added, the result should be positive. Likewise, if two negatives are added, the result must be negative. We can thus consider a check which confirms this signage, and reports an error if two numbers of the same sign incorrectly create a result of the opposite sign.



However, we must also consider the case when the two numbers being added are not the same sign. In that case, the result sign could be either positive or negative.

Input 1 Sign	Input 2 Sign	Output Sign Should Be
+	+	+
+	-	?
-	+	?
-	-	-

In order to address the case when a positive number is added with a negative number, we must first determine if any error is actually possible in these. When a positive and negative are added, the result is always between those two extreme values (it cannot be more negative or positive than the original inputs). Thus, if the inputs are both representable, all values between them are also representable and so no error is possible in those cases.

Input 1 Sign	Input 2 Sign	Error Possible?
+	+	Yes
+	-	No
-	+	No
-	-	Yes

We now need to only consider the case when the signs are the same. In that case, if the output has a different sign than the input, an error has occurred. We can first use an XNOR gate (equality) to check if both the signs are equal by taking the leftmost bits from each input (on the right, be sure to use the bit from the inverter). This result will give true if equal and false if not.

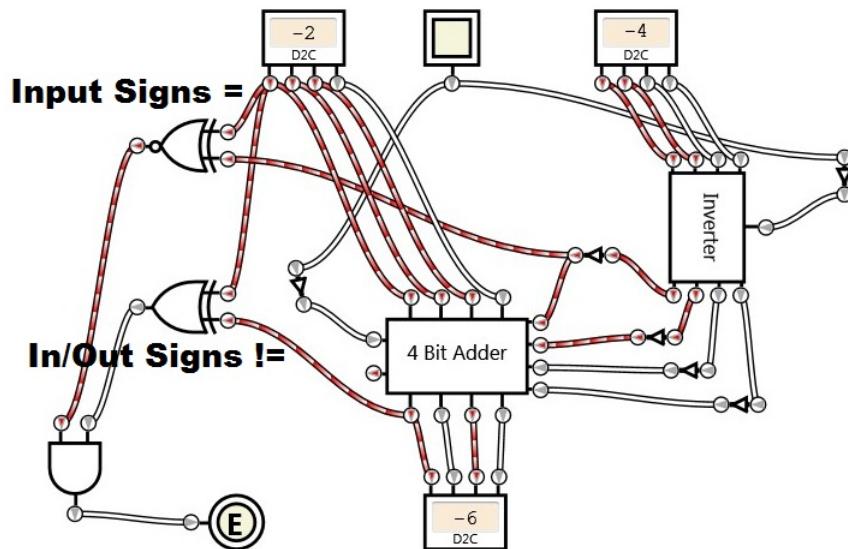
Next, we can use another XNOR gate to check if the sign bit of the output from the adder is equal to either sign bit from the input (since the input sign bits must also be equal).

With these, what decision can we make?

Input Sign Equal	Input/Output Sign Equal	Error?
T	T	F
T	F	T
F	T	F
F	F	F

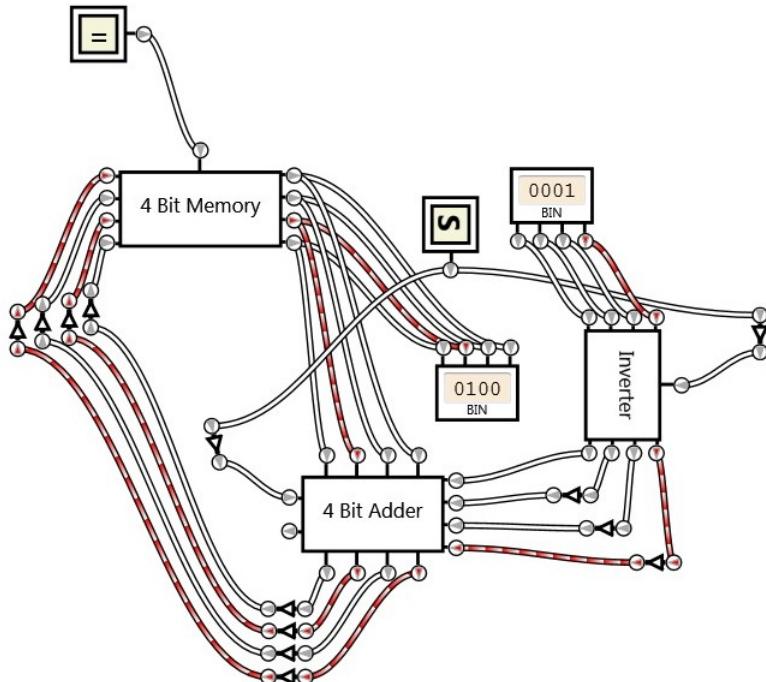
An error exists only when the input signs are equal, and the input/output signs are not equal. This condition can be established with an AND and a NOT gate on the two intermediate equalities. Finally, for input/output sign equal, note that we are taking the output of a XNOR and putting it into a NOT gate. An XNOR followed by a NOT gate is simply XOR.





8. *Problem:* Alter the four bit add/subtract circuit. Remove one of the inputs and replace it with four bits of memory. Add an accumulate button which updates the memory to be increased or decreased, as selected, by the amount of the input.

Solution: We will use the four bit memory defined previously in this chapter. In order to construct an accumulator, the output of the adder must become the input to the memory. The output from the memory then goes back into the adder. In addition, the final numeric output must now be read from the memory instead of the adder, to avoid fluctuations as the remaining input value is changed.

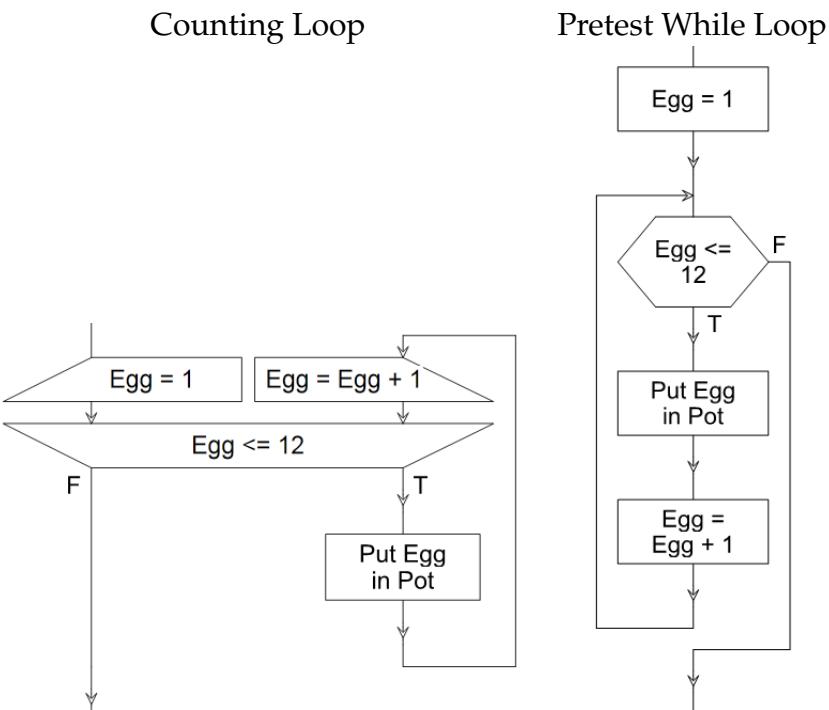


A.17 Flowcharts

Exercises found in Chapter 19 on page 222.

1. *Problem:* Convert the 12 eggs counting loop example into pretest while loop style.

Solution: The order of steps will precisely follow the order shown in the original counting loop. First, the initialization block must be executed (setting Egg = 1). Next, the loop condition will check to see if egg is in the range (12 or less). The loop body will then execute, and finally the egg counter will be updated. The loop condition will then be checked again. Compare the original counting loop with the pretest while loop style and note the direct correspondence.



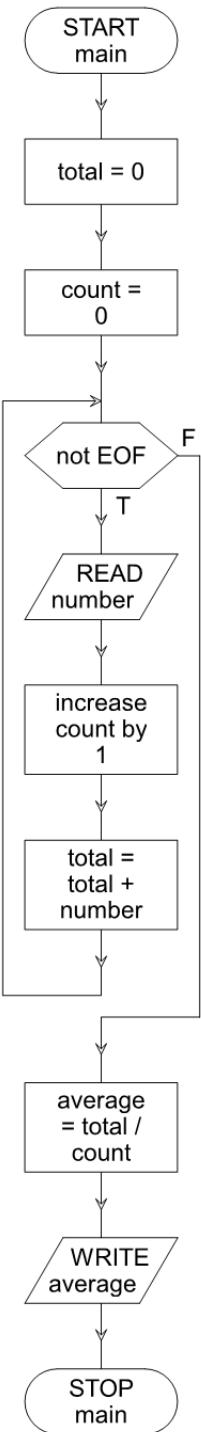
2. *Problem:* Write a flowchart that calculates the average (mean) of a list of numbers.

Solution: The average (mean) is defined as the sum divided by the number of elements. We will use both a counter and an accumulator, and read the values one at a time from the input. When each value is read, it will be added to the total (the accumulator) and the count will be increased by 1. When there are no more values, the total will contain the sum and the count will indicate the number of values. A division can then be used to find the mean.

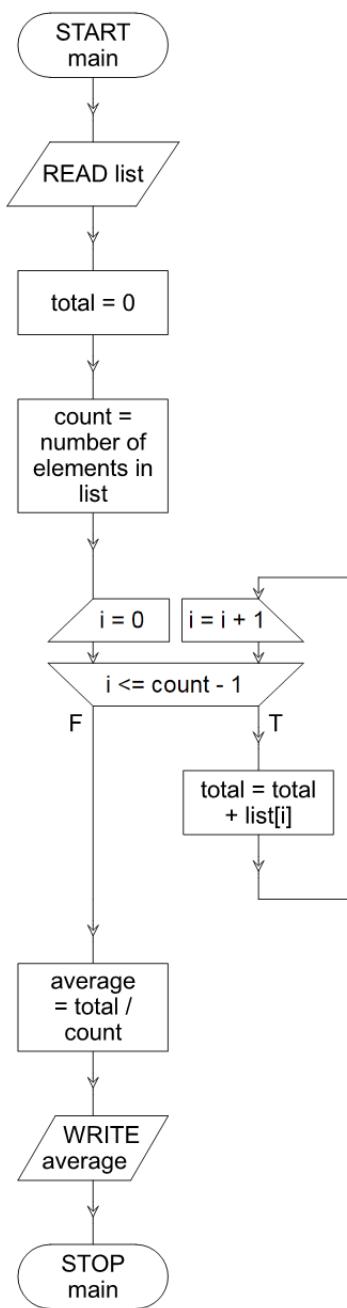
Alternative Solution: We will input the entire list up front and use a counting loop to traverse it. Each element in the list will be accessed by index (with index 0 being the first element) and accumulated into a total.



One Value at a Time



Whole List Up-Front

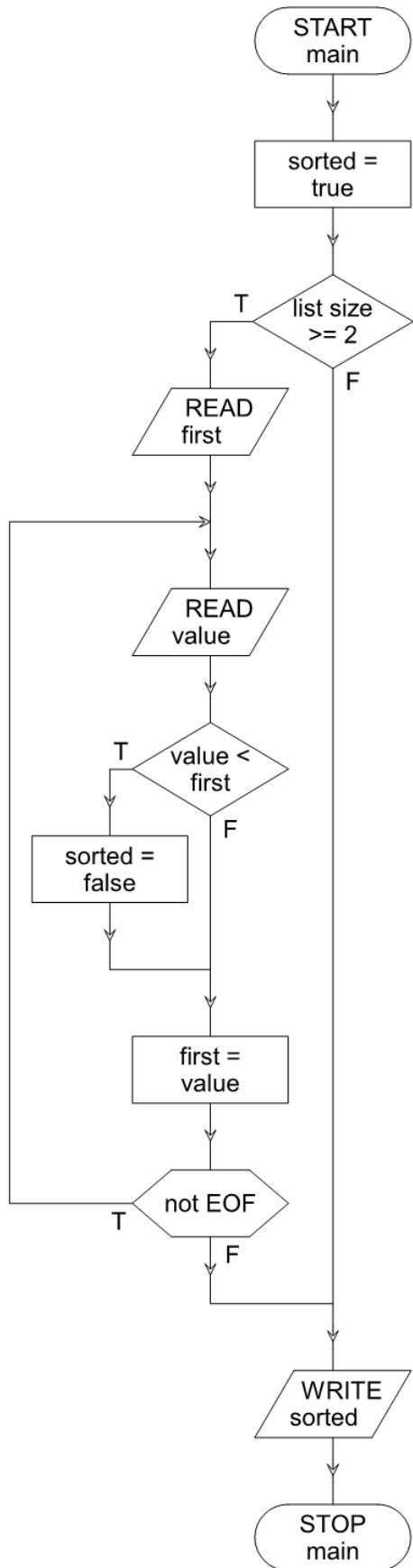


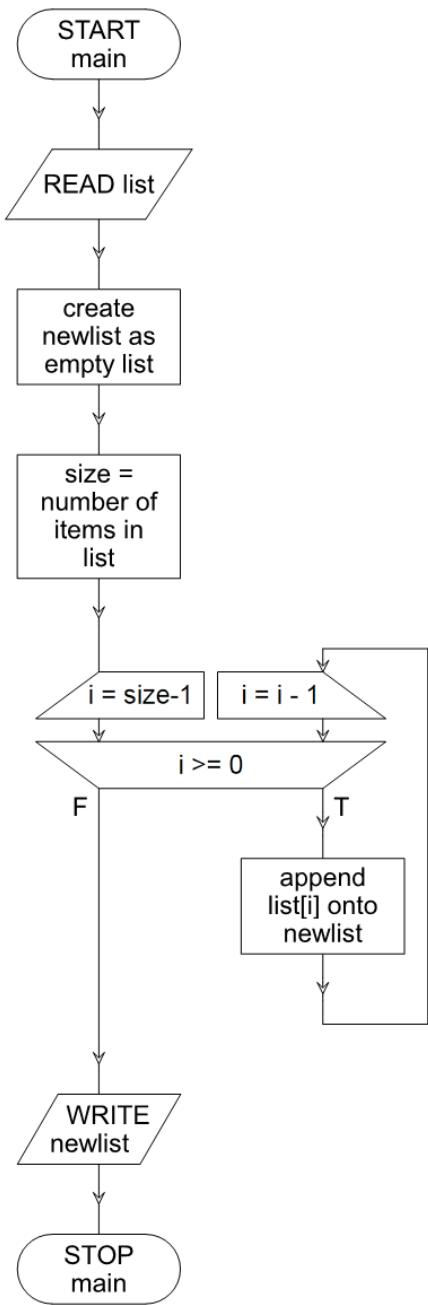
3. *Problem:* Write a flowchart that determines whether or not a given list of numbers is in sorted order from smallest to largest.

Solution: It is not necessary to create a nested loop and compare every item to every other item. Instead, if each adjacent pair of items is in sorted order, then the entire list is in sorted order. Consider a list which has items a , b , and c in that order. If $a \leq b$ and $b \leq c$, then it must also be true that $a \leq c$. This property (known as transitivity) continues to hold as the size of the list increases.

We will start by assuming the list is sorted; and if we find even one case where items are out of order, we will conclude it is NOT sorted. We will read an initial value and then a next value (because we need two values to compare). If the later value is less than the initial value, then the list is not in sorted order. As long as more items exist, we compare the next pair: the current value becomes the initial value, and a new next value is read in.

There is also a special case to consider: in order for the above procedure to work, the list must contain at least two elements. What about a list that contains no elements, or only one element? In these cases, the list must be sorted. There is no way for a list to be out of sorted order if it contains less than two elements. So if there are less than two elements, we skip the whole procedure.





4. *Problem:* Write a flowchart that, given a list, outputs the list in reverse.

Solution: For example, assume the input list is 1, 3, 5, 4, then the output list will be 4, 5, 3, 1. To accomplish this, we will read in the entire list up front. Then, using a counting loop, we will traverse the list backwards (starting at the end and going to the beginning). At each step, the value will be appended onto a new list, so that the last value in the original list is the first value placed on the new list, and so on.

Be sure to note the changes to the normal counting loop: the loop index is initialized to $\text{size} - 1$ instead of 0 (the minus 1 is because indices start at 0, so the if the list has 3 elements, the last index is 2). At each iteration, this index value is decreased by 1 (instead of increased) to move toward the beginning of the list. After index 0 is processed, the loop exited.

5. *Problem:* The Fibonacci sequence is a list of numbers, where each number is derived by adding the previous two together. The first two numbers in the sequence are 1. The first eight Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21.

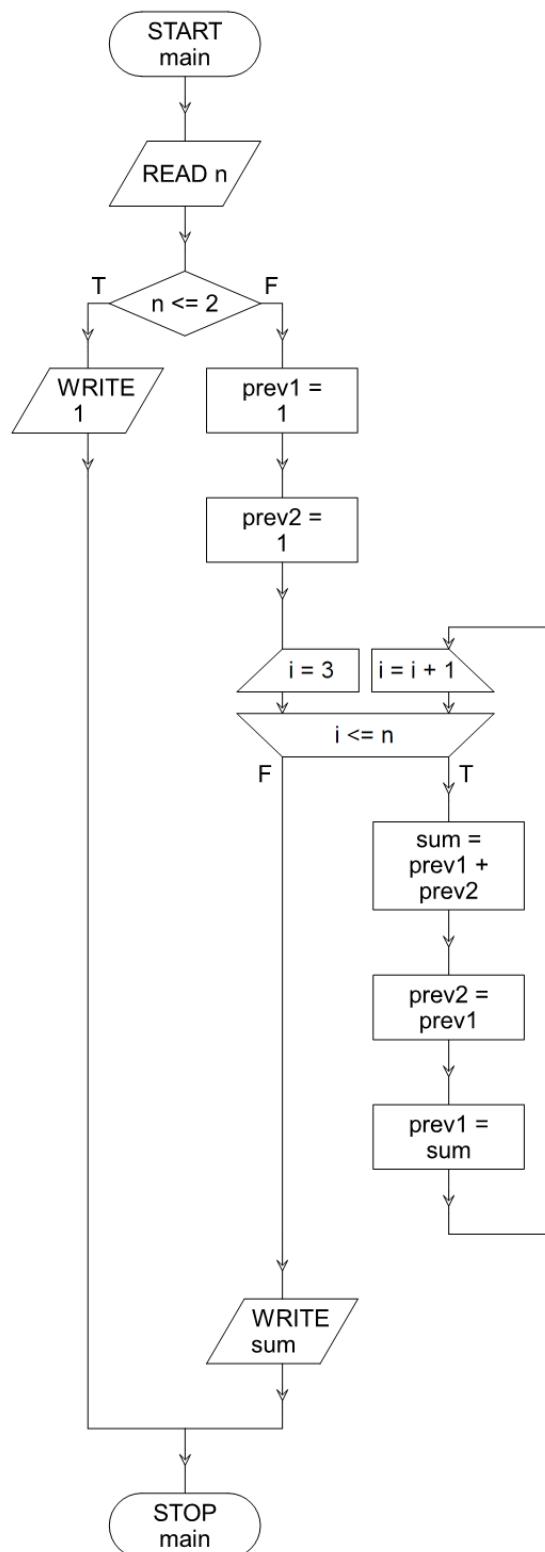
Write a flowchart that given a position number, outputs the corresponding Fibonacci number. For example, if the position is 1 or 2, the Fibonacci number is 1. If the position is 6, the Fibonacci number is 8, and so on.

Solution: Given that each Fibonacci is defined as the sum of the previous two, we can compute the sequence incrementally, always maintaining a record of the two most recent Fibonacci numbers. When we reach the position desired, we stop computing and output that value.

The first two numbers themselves can be considered special cases; if the desired position is 1 or 2, we will simply output 1. For other cases, a loop can be used. We start the third position as being based on the previous two values (both 1). Then, at each loop iteration, a new sum is computed by adding the previous two together. We then put the computed sum as the most recent previous Fibonacci number and the loop repeats.

In order to manage how many times the loop should repeat, a counting loop based on the desired position can be used. Note that the initial value is not 0, but 3, since the first iteration through the loop calculates the third position (if the previous two values are initialized to 1).

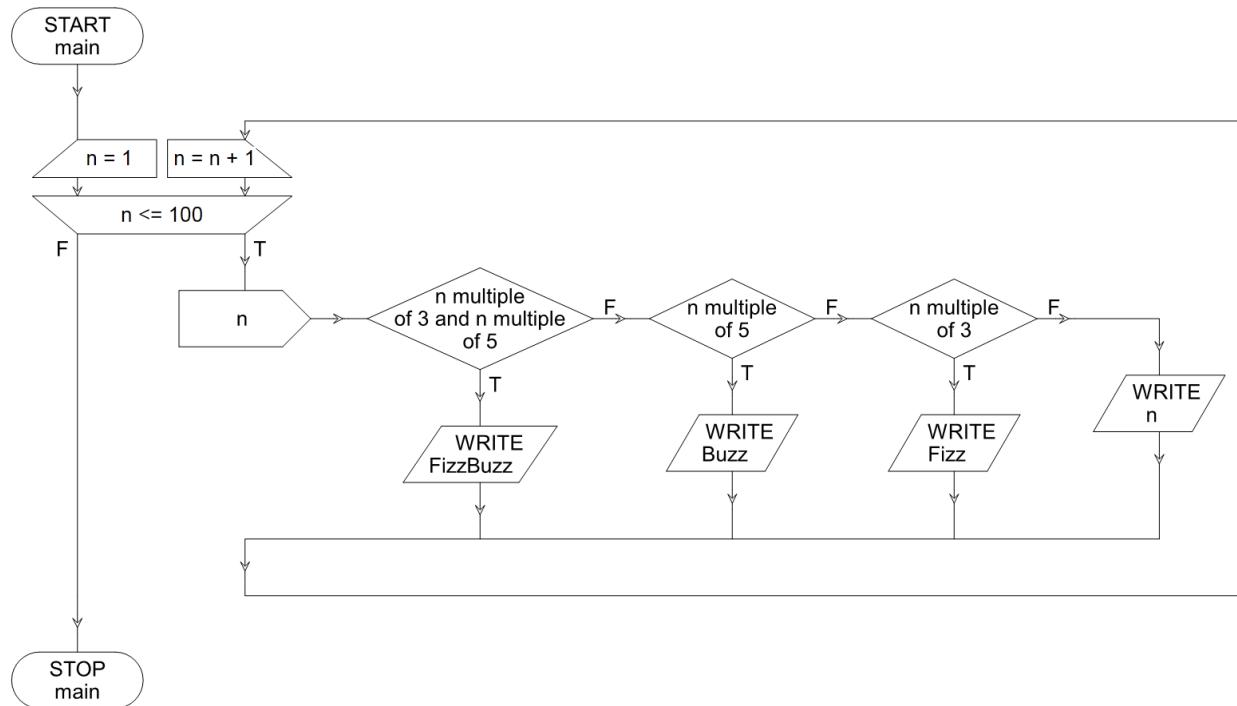
The order in which the previous values are stored is significant: in order to avoid duplicating or losing a value, we must first shift the “one step old” value into the “two step old” position, and then shift the most recent value into the “one step old” position.



6. *Problem:* Consider the Fizz-Buzz problem, a small programming challenge proposed for use in job interviews. Write a flowchart that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

Solution: We need to loop, using a counting loop, through all values in the range of 1 up through 100. If the number is a multiple of three or five or both, then a special message is output. Otherwise just the number is output.

For each value, we first check if it is a multiple of both 3 and 5 (this can be determined by checking if the number is evenly divisible by 3 and 5, a condition easily checked by the computer). If it is a multiple of both, we output "FizzBuzz". Only if it is not a multiple of both do we check the individual conditions. Finally, if the number is not a multiple of interest, then we simply output the number itself.

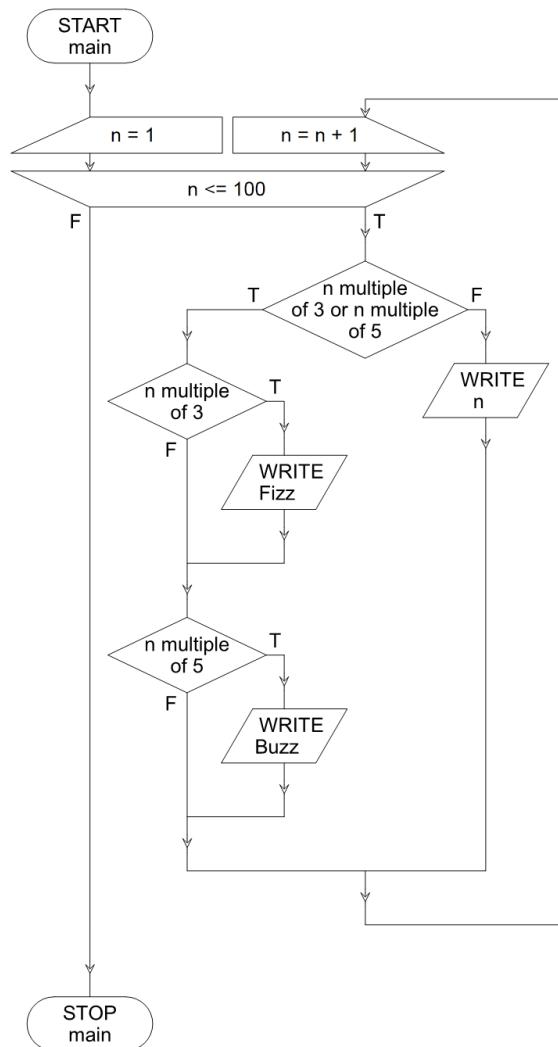


Note: to say a number is a multiple of both three and five means that it is multiple of 15 (because $3 * 5 = 15$). Therefore, the condition that the number is a multiple of both three and five could be replaced with a single condition that the number is a multiple of 15.

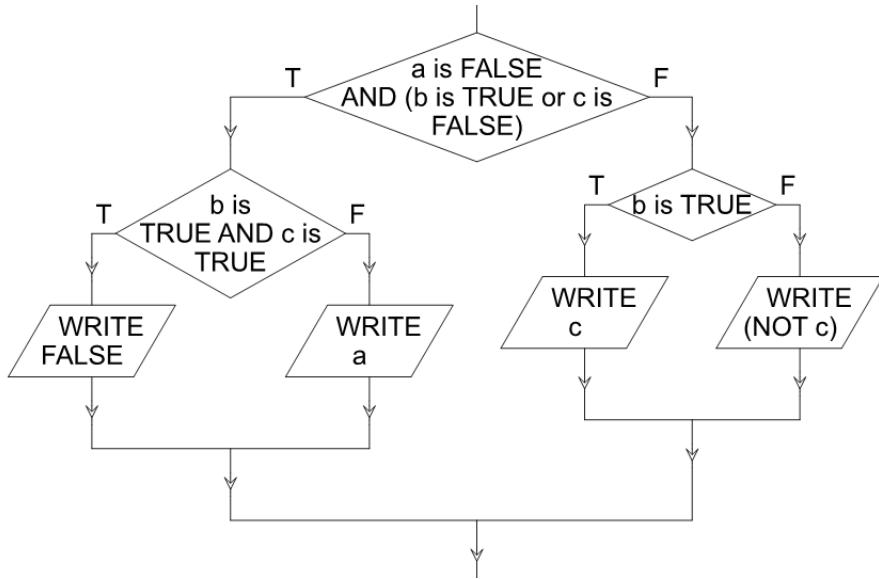
Alternative Solution: Rather than having three different outputs, we can note that the “both” case is a composite of the two individual cases (3 = Fizz, 5 = Buzz, both = FizzBuzz). Thus, we can sequence two if blocks, one for multiple of three, and one for multiple of five, and have each output the respective portion. If the number is a multiple of both three and five, both blocks will be run and so the total message FizzBuzz will be output.

The only possible difficulty is ensuring that the number is output only when appropriate; we can't do an else on either of the if statements, as the other may still be true. Thus, we need to wrap the entire Fizz/Buzz generator in an if to see if we should even try the multiples or not.

If the number is a multiple of 3, or a multiple of 5 (or both), then we know it is ok to check for Fizz and/or Buzz. However, if the number is not any of those multiples, we skip to outputting the number itself.



7. Problem: Assume a , b and c are Boolean variables; determine a logical expression for the following flowchart.



Solution: The easiest way to create an expression from a complex flowchart such as this one is to first generate a truth table, then convert the truth table into an expression.

First, start with an empty truth table. Note that there are three variables in the flowchart so the truth table will also have three input variables, and thus eight rows.

a	b	c	result
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

Given that a claim like “ a is false” is the same as $\neg a$, the first decision can be rendered as $\neg a \wedge (b \vee \neg c)$. All of the rows where this expression is true will be determined by the left side of the flowchart; all of the rows where this expression is false will be determined by the right side of the flowchart.

Start by marking all rows which meet the initial expression. The value of these marked rows we will determine by analyzing the left side of the flowchart.



a	b	c	result
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

The left side of the flowchart gives us the expression $b \wedge c$. Are there any rows highlighted which meet this criteria? If so, according to the flowchart, they have the output of false. Of the three marked rows, the first one meets the criteria (it has a as false, b as true, and c as true). The remaining two marked rows will have the output of a . In both remaining cases, we will copy the value of a in each row into the result.

a	b	c	result
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	F
F	T	F	F
F	F	T	
F	F	F	F

What of the other, unmarked rows? They must be solved using the right side of the flowchart. The value of these rows will be decided on the basis of b . First, select those remaining rows in which b is true (do not update already completed rows: they are not subject to this portion of the flowchart).

a	b	c	result
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	F
F	T	F	F
F	F	T	
F	F	F	F

According to the flowchart right portion, when b is true, the result output is the value of c . So in the highlighted rows, we will simply copy the value of c into the result.



a	b	c	result
T	T	T	T
T	T	F	F
T	F	T	
T	F	F	
F	T	T	F
F	T	F	F
F	F	T	
F	F	F	F

Finally, in the remaining rows, the flowchart indicates the value $\neg c$ is the result. So we will write the opposite of c into result for each remaining row (that is, if c is true, we write false, and vice versa).

a	b	c	result
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

Given this complete truth table, we can find a logical expression using disjunctive normal form and simply extracting the true rows. If desired, the expression could be simplified using the Boolean simplification rules.

The resulting expression (unimplified) is $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge \neg c)$.

Alternative Solution: Build an expression based on each side of a decision. For each decision, represent the condition with some variable v . Then, construct an expression L for the true side of the decision and an expression R for the false side of the decision. Combine these with the total expression $(v \wedge L) \vee (\neg v \wedge R)$. Substitute in the definitions of v , L , and R .

To apply this technique, the root expression $v = \neg a \wedge (b \vee \neg c)$. To find the left expression, perform the technique recursively. The left decision condition is $v_L = b \wedge c$. The two branches of this left decision can be combined, as shown above, to form $(v_L \wedge L_L) \vee (\neg v_L \wedge L_R)$. In this case, $L_L = F$ and $L_R = a$. Thus, the left side of the flowchart may be represented as $L = ((b \wedge c) \wedge F) \vee (\neg(b \wedge c) \wedge a)$.

Likewise, to find the right expression, we will note the right decision condition is $v_R = b$, with $R_L = c$ and $R_R = \neg c$. As above, these combine to form $R = (b \wedge c) \vee (\neg b \wedge \neg c)$.

With these sub-expressions, we can assemble the final expression using the original pattern and the values for v , L , and R that have been discovered. The complete expression is $((\neg a \wedge (b \vee \neg c)) \wedge (((b \wedge c) \wedge F) \vee (\neg(b \wedge c) \wedge a))) \vee (\neg(\neg a \wedge (b \vee \neg c)) \wedge ((b \wedge c) \vee (\neg b \wedge \neg c)))$.

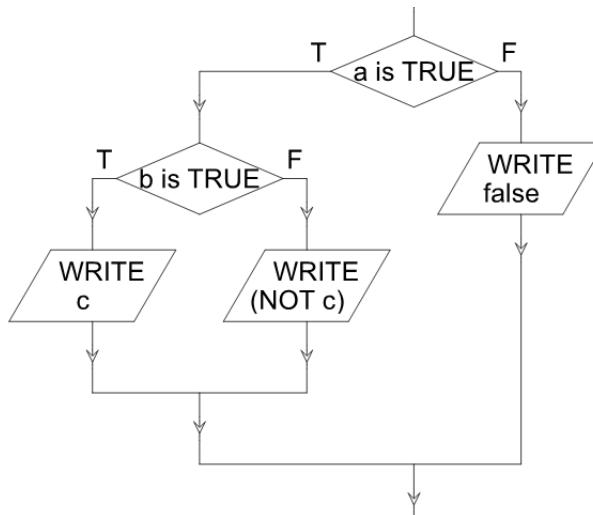


This expression could be simplified using Boolean simplification rules, or by constructing a truth table and extracting an expression from it.

8. *Problem:* Create a step-by-step version of the flowchart from the previous problem, using only one variable per decision condition (in other words, no AND or OR).

Solution: Start with a Boolean expression representing the output of the previous flowchart. As shown in the previous solution, the expression $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge \neg c)$ is equivalent to the flowchart's output. Notice that both cases require a to be true; that is, by the distributive law, we could rewrite this expression as $a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))$.

Start by splitting on the value of a ; if a is false, then the output is false. If a is true, however, then we must consider our next step. Given that a is true, then the output will be true if b and c are equal (both true, or both false). We can save some space by taking advantage of this relationship. We can evaluate one of the variables (in this case we'll choose b but there is no significance to that choice), and if it is true, output the value of the other variable (c). On the other hand, if b is false, we can output $\neg c$. This establishes the $b \wedge c$ or $\neg b \wedge \neg c$ options without another level of nested decisions.

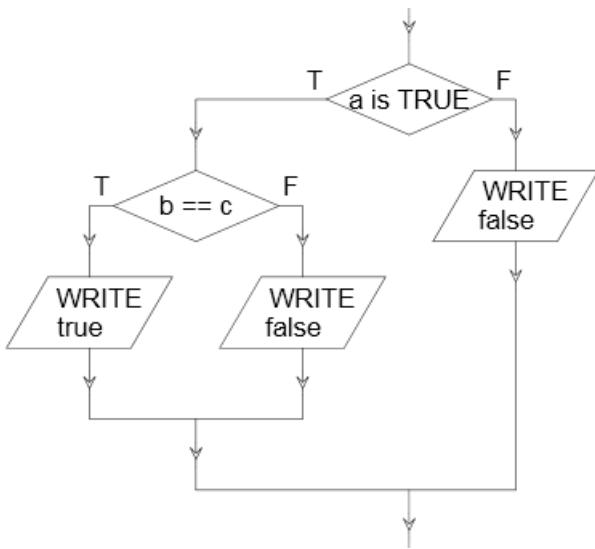


Alternative Solution: Begin as before. However, given the expression $a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))$, another transformation is possible (alluded to in the previous solution).

1. $a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))$	Initial Expression
2. $a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))$	DeMorgan's Law
3. $a \wedge \underline{(b \wedge c) \vee \neg(b \vee c)}$	Double Negation
4. $a \wedge \neg\neg((b \wedge c) \vee \neg(b \vee c))$	DeMorgan's Law
5. $a \wedge \neg(\neg(b \wedge c) \wedge \neg\neg(b \vee c))$	Double Negation
6. $a \wedge \neg(\neg(b \wedge c) \wedge (b \vee c))$	Definition of exclusive OR
7. $a \wedge \neg(b \oplus c)$	Definition of equality
8. $a \wedge (b \leftrightarrow c)$	Final Expression

Now the relationship $b \leftrightarrow c$ is explicit and may be embedded directly into the flowchart.





A.18 Analysis of Algorithms

Exercises found in Chapter 20 on page 236.

1. *Problem:* Two algorithms have been created to process input records. The first algorithm processes 100 records in 13 seconds. On the same computer, the second algorithm processes 100 records in only 8 seconds. Which algorithm will perform faster on 200 records?

Solution: Unable to tell. A single data point is not enough information to determine the algorithm's time complexity. At least two data points are needed to distinguish constant from linear time algorithms, and more are needed to distinguish the other complexity classes.

2. *Problem:* One algorithm was step-counted and found to complete in $4n + n^2 + \log n$ steps (with input size n). A second algorithm was step-counted and found to complete in $2n^3 + 4$ steps (with input size n). Which algorithm has the better time complexity?

Solution: In each case, the time complexity of an algorithm is determined by the most significant term. In the equation $4n + n^2 + \log n$, the most significant term is n^2 , so the first algorithm is $O(n^2)$. In the equation $2n^3 + 4$, the most significant term is $2n^3$. Coefficients are dropped in time complexity, so this algorithm is $O(n^3)$. The first algorithm has the better time complexity.

3. *Problem:* Find the time complexity of the following algorithm:

- (a) Input a and b as positive whole numbers
- (b) Let n be the larger of a and b
- (c) If $a \div n$ is a whole number, AND
- (d) If $b \div n$ is a whole number, return n as the answer.
- (e) Otherwise, modify $n = n - 1$
- (f) Go to line (b)

Solution: We will use the loop analysis technique. Repetition exists in lines (b) through (f), which repeat until certain conditions are met. Eventually, as n is decremented, the condition $a \div 1$ and $b \div 1$ will be reached when n reaches 1. This is true because for any positive whole number, if it is divided by 1, the result will be itself, a positive whole number, which meets the algorithm's check criteria. How many times will this sequence of steps (b) through (f) repeat? It may stop at any time, but in the worst case (we are always interested in the worst case for our analysis), n might have to travel all the way down to 1. Due to the fact that n decreases by 1 at each step, this could result in up to n total runs through the loop. Thus, this algorithm is $O(n)$, linear time.



4. *Problem:* Find the time complexity of Euclid's Algorithm.

Solution: Recall Euclid's Algorithm:

- (a) Input a and b as positive whole numbers
- (b) If b equals 0, return the answer a
- (c) Save b in a temporary variable t
- (d) Update b to be the remainder of $a \div b$
- (e) Update a from t
- (f) Go to line (b)

The loop analysis technique works for this example as well. Note that the repetition exists in lines (b) through (f). This repetition is based primarily on b (reaching zero will end the loop). Each iteration modifies both a and b , and, in a sense, swaps them, as a becomes the previous value of b , and the new value of b is derived from a using the remainder of a division.

The largest remainder will occur when b is large and a is $b - 1$. In this case, however, a takes on the previous value of b and the algorithm ends in the next step. In other cases, some additional steps are needed, but it seems like the input is not being consumed by a fixed amount (as in the previous problem) but is being consumed by division, getting quickly smaller at each step. An algorithm which divides the input at each step is usually logarithmic time $O(\log n)$. There is a problem, however. There is no n defined in this problem, so we must be careful to express the time complexity in terms of the actual input.

We are not attempting to be precise with a tight upper bound, so simply noting that both inputs have the same division effect is sufficient. We'll call it $O(\log(ab))$.

(Note: detailed mathematical studies of this version of Euclid's algorithm have proven that the worst-case time complexity is $O(\log_{10} \min(a, b))$.)

Both this algorithm and the algorithm in the previous problem find the greatest common factor of two numbers. This version of Euclid's algorithm, however, has a better time complexity.

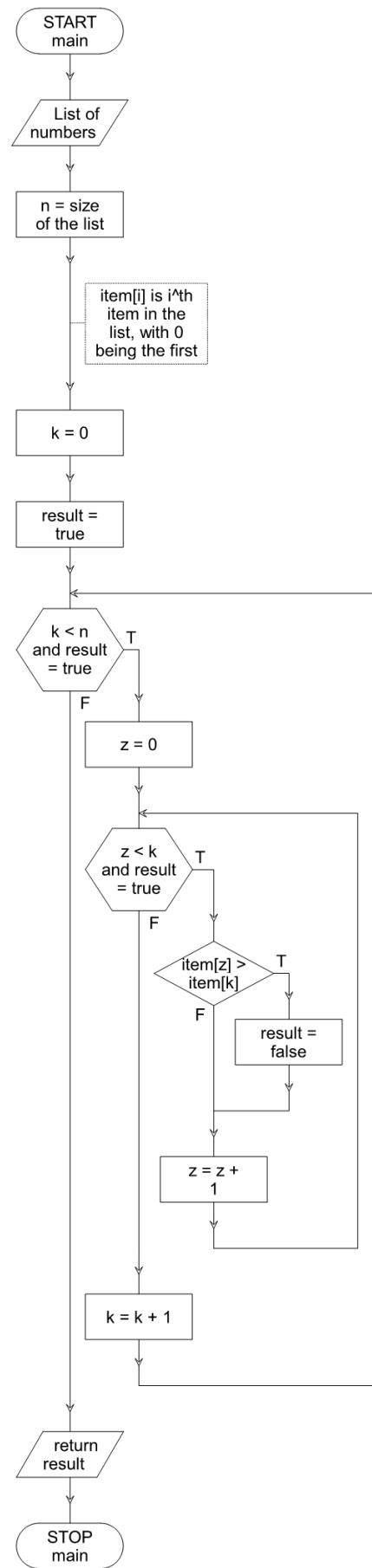


5. *Problem:* Find the time complexity of the following algorithm:

- (a) Let $item_i$ represent the i^{th} item in a list of n numbers, with $item_0$ being the first element.
- (b) Let $k = 0$
- (c) If $k = n$ then return true
- (d) Let $z = 0$
- (e) If $z = k$ then go to line (i)
- (f) If $item_z > item_k$ then return false
- (g) Update $z = z + 1$
- (h) Go to line (e)
- (i) Update $k = k + 1$
- (j) Go to line (c)

Solution: This algorithm has a lot of “go to’s! In order to get a better grip on what is happening, transform the algorithm into a flowchart. This example was shown previously in the chapter on flowcharts. The resulting flowchart is reproduced here.

From the flowchart, it is possible to see that the algorithm consists of a nested loop pair. The outer loop, based on k , increments by a constant and loops up to the size of the input, so this outer loop is linear. The inner loop, based on z , increments by a constant, but only loops up to k . How can we handle this? Although k is not the input size, it increments up to the input size, so we can consider z to be based on the input as well. Nested loops, both incrementing a constant amount and both based on the input size, gives a time complexity of $O(n^2)$, quadratic time. Note that the decision in the heart of the loops is itself not a loop, so this does not count as a triple-nested loop.



6. *Problem:* Devise an alternative, more efficient, algorithm which computes the same result as the algorithm given in the previous problem. Find the time complexity of the improved algorithm.

Solution: In order to solve this problem, a detailed understanding of the previous algorithm is needed: not just how it is structured, but *what* it is trying to accomplish. The output of the algorithm is a single true/false variable, result. This result is treated as a flag: it starts true, but once turned false, it stays false. So the algorithm is looking for the presence of a certain condition. The result is true if that condition is not found, or false if the condition is found. What condition? The condition $item_z > item_k$. Looking at the loop for z and k , notice that the loop condition is $z < k$.

Therefore, this algorithm is searching for any pair of items in the list, where the earlier item is greater than the latter item. In other words, this algorithm is checking if a list is in sorted order or not. The result is true if the list is sorted; the result is false if the list is not sorted. The algorithm makes no attempt to actually sort the list.

In order to create a better algorithm for this task (determining if a list is sorted), we can note that it is not necessary to compare all pairs (which the previous algorithm does). Instead, we need only compare adjacent elements. If a list is not in sorted order, there will be some adjacent elements which themselves are out of order. This can be done in a single loop, considering each item and its successor.

- (a) Let $item_i$ represent the i^{th} item in a list of n numbers, with $item_0$ being the first element.
- (b) Let $k = 0$
- (c) If $k \geq n - 1$ then return true
- (d) If $item_k > item_{k+1}$ then return false
- (e) Update $k = k + 1$
- (f) Go to line (c)

To find the time complexity of this algorithm, examine its loop. There is a single loop, from items (c) through (f). This loop is based on the input (actually it goes one less than the size of the input, but constant adjustments of this nature do not affect the complexity result). The loop increments a constant amount at each step. Therefore, this algorithm is $O(n)$, linear time. That is a better time complexity than the previous algorithm.

7. *Problem:* Your company has a records processing algorithm which runs overnight to process the day's sales. When the algorithm was first implemented, there were about 20 sales a day, and the algorithm took about five minutes to run. A few months later, daily sales averaged about 100 per day, and you notice the algorithm is now taking about two hours to run. The newly hired sales manager claims she can triple the company's sales. If she does, will the algorithm still finish in time for business open at 8:00AM if it is started at 5:00PM when the business closes?



Solution: First, identify the time complexity of the algorithm. Only two data points make time complexity identification an inexact art; however, we can assume that there is limited constant overhead.

If the algorithm were linear time, and took 5 minutes to process 20 sales, then we would expect 10 minutes to process 40 sales, 20 minutes to process 80 sales, and 40 minutes to process 160 sales. The algorithm is taking much more time than that, so it is probably worse than linear time.

On the other side, consider exponential time. Exponential time algorithms take twice as long for each additional element. So if the algorithm took 5 minutes to process 20 sales, then we would expect 10 minutes to process 21 sales, 20 minutes to process 22 sales, 40 minutes to process 23 sales, and so on. A reasonable amount of extrapolation shows that if the algorithm were exponential, there is no way it would be completing 100 records in our lifetime.

So the algorithm must be better than exponential. This leaves the polynomial times. If the algorithm were quadratic, then a doubling of input size would result in about four times longer to process. Thus, if the algorithm took 5 minutes to process 20 sales, then we would expect 20 minutes to process 40 sales, 80 minutes (1.3 hours) to process 80 sales, and 320 minutes (5.3 hours) to process 160 sales.

The actual data shows that 100 sales took 2 hours, suggesting quadratic $O(n^2)$ is likely the algorithm's time complexity. We can check this result by noting the factor of difference of the two sample inputs (number of sales) is 5 ($20 * 5 = 100$), and therefore the time difference should be approximately $5^2 = 25$ times. Indeed, $5 * 25 = 125$, very close to the 120 minutes measured.

Following this reasoning, if the sales volume is tripled to 300, we expect about $3^2 = 9$ times longer will be required. $120 * 9 = 1080$ minutes, or 18 hours will be required. This time exceeds the available time between close and open the next business day.

8. *Problem:* Given the previous solution, Mike the IT guy notes that all sales are currently being processed on one server. He proposes buying several more servers (which perform at the same speed as the current server) to distribute the load.
 - (a) Assuming the load can be equally distributed between servers, how many servers would be required to complete the job in time?
 - (b) If the total sales per day increases to 400, how many servers would be required to complete the job in time?

Solution: We will assume that 300 sales will take 18 hours to process on one server. The actual time allotted is from 5:00PM to 8:00AM the next day: 15 hours. If the company buys just one more server, each of the two servers can work for a little over 9 hours each night to complete the task.

To determine the load requirement if the total sales increased to 400, first note that $\frac{400}{300} \approx$



1.3. Applying quadratic time formula, $1.3^2 \approx 1.8$, so the total time required is about 32 hours; a total of three servers would be required.



Bibliography

Many excellent sources were consulted in the writing of this text, and further study of any or all of the topics therein is recommended to the interested student.

The ★ symbol indicates a source highly recommended.

B.1 Wikipedia

Although the practice of citing Wikipedia as a source is often frowned upon, the fact remains that Wikipedia is an excellent source of general information, especially relating to technical fields. Errors have, on occasion, been found (by myself included) and certainly are occasionally intentionally introduced by vandals. However, the value of Wikipedia in its broad and detailed coverage of every topic in this book cannot be overlooked.

Glossary words make excellent starting searches on Wikipedia is more detail if desired.

If concern about the validity of statement on Wikipedia is raised, check the edit history to see if the statement is a recent addition or long-standing. Long standing statements are more agreed upon by readers and editors. Also worth checking is the topic's discussion page, where debate over the validity of claims, and possible concerns of errors, are voiced. Finally, most computer and mathematical oriented claims can be directly tested. If unsure about the validity of a claim, devise a test to determine if it works.

B.2 Websites

Anderson, Sean E. "Bit Twiddling Hacks" Retrieved from
<http://graphics.stanford.edu/~seander/bithacks.html> A collection of bitwise operations that quickly calculate a wide variety of results.

★ Azad, Kalid. "Easy Permutations and Combinations" Retrieved from
<http://betterexplained.com/articles/easy-permutations-and-combinations/> An easy to follow discussion of how permutations and combinations work.

Bigelow, Ken. "Digital Logic" Retrieved from
<http://www.play-hookey.com/digital/>
As the name implies, this website is a collection of circuits which can be built from basic logic gates.

Finley, Thomas. "Two's Complement" Retrieved from
<http://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html>
Describes how to perform Two's Complement, and includes an extra section "Why Inversion and Adding One Works"

Goldberg, David. "What Every Computer Scientist Should Know About Floating-Point Arithmetic" Retrieved from
http://docs.sun.com/source/806-3568/ncg_goldberg.html
An in-depth analysis of binary floating point and its limitations.

Hollasch, Steve. "IEEE Standard 754 Floating Point Numbers" Retrieved from
<http://steve.hollasch.net/cgindex/coding/ieeefloat.html>
A good discussion of floating point without excessive detail.

Leon, Jeffrey. "Hadamard Matrices and Hadamard Codes" Retrieved from
http://www.math.uic.edu/~leon/mcs425-s08/handouts/Hadamard_codes.pdf
Contains various Hadamard matrices and discusses their application in error detection and correction.

★ Koehler, Kenneth. "Elementary Computer Mathematics" Retrieved from
<http://www.rwc.uc.edu/koehler/comath/toc.html>
This excellent online textbook emphasizes number systems, signed, unsigned, and floating point numbers; logical operators and truth tables, with some discussion of logic circuits, set theory, and finite state machines. Exercises are provided.

★ Martin, David. "Sorting Algorithm Animations" Retrieved from
<http://www.sorting-algorithms.com/> and child pages
In addition to a variety of visualizations, contains a description of the algorithm, time and space complexity for many common sorting algorithms.

★ Mastascusa, E. J. "Digital Signals and Logic" Retrieved from

http://www.facstaff.bucknell.edu/mastascu/elessonshtml/TOC_BitsBytes.html

Contains excellent resources on logic gates, digital logic, and data representation. Many fully worked exercises also included.

Permadi, F. "Converting Decimal to Hexadecimal" Retrieved from
<http://www.permadi.com/tutorial/numDecToHex/>

Permadi, F. "Converting Hexadecimal to Decimal" Retrieved from
<http://www.permadi.com/tutorial/numHexToDec/>

Pigeon, S. "Huffman Codes" Retrieved from
<http://hbfs.wordpress.com/2011/05/17/huffman-codes/>

Seguin, Karl. "Simple Algorithms" Retrieved from
<http://algorithms.openmymind.net/>

★ Spolsky, Joel. "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" Retrieved from
<http://www.joelonsoftware.com/articles/Unicode.html>

The Unicode Consortium. "The Unicode Standard" Retrieved from
<http://www.unicode.org/standard/standard.html> and child pages

Wagner, Neal. "The Laws of Cryptography" Retrieved from
<http://www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf>
Includes Hamming Code and general information on channels and information theory.

Wright and Rebelsky. "The Binary System" Retrieved from
<http://www.cs.grinnell.edu/~rebelsky/Courses/CS152/97F/Readings/student-binary.html>
Discussion of binary representation, addition, multiplication, and division.

Young, James F. "Going With the Flow" Retrieved from
<http://www.clear.rice.edu/elec201/ictutorial/tut3-1.html>
Tutorial sections 3 and 4 describe how flowcharts and control structures work in the context of the C language.

Zanden, Brad. "Analysis of Algorithms" Retrieved from
<http://www.cs.utk.edu/~bvz/bvz/classes/cs302/notes/complexity.html>
Contains an example problem that falls into each of the common complexity classes.

B.3 Books

Cormen, Thomas, et al. *Introduction to Algorithms* MIT Press 2003. ISBN 0262033844

Dewdney, A. K. *The New Turing Omnibus: 66 Excursions in Computer Science* Henry Holt 1993. ISBN 0-8050-7166-0

Farrell, Joyce. *Programming Logic and Design* Fourth edition. Course Technology 2006. ISBN 1-4188-3634-6

Harel, David. *Algorithmics: The Spirit of Computing* Addison Wesley 2004. ISBN 0321117840

Hoffer, George, Valacich. *Modern Systems Analysis and Design* Fourth edition. Pearson Prentice Hall 2005. ISBN 0-13-145466-8

Lipschutz, Seymour. *Schaum's Outline of Essential Computer Mathematics* McGraw-Hill 1982. ISBN 0070379904

Rood, Harold. *Logic and Structured Design for Computer Programmers* Third edition. Brooks/Cole 2001. ISBN 0-534-37386-0

★ Petzold, Charles. *Code: The Hidden Language of Computer Hardware and Software* Microsoft Press 2000. ISBN 0-7356-0505-X



Glossary

A

Accumulator

a variable which stores intermediate results of an aggregation, usually a sum, 212

Additive Counting Rule

When selecting just one item from several sets, the total number of possibilities is the sum of the cardinalities of the sets, 9

Algorithm

a step-by-step process to solve a certain problem or class of problems, 198

Alpha

a component of a color indicating how translucent it is, 141

Alpha Blending

mixing a translucent color over a background to determine the actual color to display at that location, 141

ASCII

short for American Standard Code for Information Interchange, a 7 or 8 bit encoding of English letters, digits, punctuation, and other symbols, 123

Associative

a function that, in a sequence of that function, the arrangement of parentheses can be changed without changing the final result, 28

B

Base

number of symbols or digits in a particular number system, 91

Berger Code

error detecting code capable of detecting any number of errors as long as all errors are of one type (such as 1 to 0), 159

Bias

the amount a stored value is offset from its actual value, 114

Big-Oh

short for “biggest order”, indicates the most significant complexity term, with coefficients dropped, 225

Binary

base 2 number system with digits 0 and 1, 91

Binary Operator

an operator that takes two inputs, 39

Binary Search

a search technique for ordered lists which cuts the search space in half at each step, 227

Binomial Coefficient

Formally, coefficient of the x^r term in the polynomial expansion of $(1+x)^n$. Practically, the number of ways to select r items from a set of n , 10

Bit

binary digit, a single number with the value 0 or 1, 91

Bit Mask

a series of bits that are manipulated using bitwise operations, usually to define portions of number as usable or not, 149

Boolean

a 2-valued object, whose values are usually represented as yes/no, true/false, or 1/0, 37

Boolean Flag	a Boolean variable which starts out at one value (usually false) and switches value in one direction only (usually to true) under a variety of conditions, 57
Burst Error	a contiguous sequence of bits all of which are incorrect, 156

C

Cardinality	the number of elements in a set, 4
Character Encoding	a transformation which describes how to store a particular code point in memory or on disk, 125
Clock	a regular pulse signal which synchronizes the activities of logic circuits throughout a component, 192
Code Point	a number which represents a specific symbol in Unicode, 125
Code Rate	a measure of the overhead of an error detection or correction code, 156
Color Depth	bits per pixel in an image, 138
Color Model	a technique for representing colors using a collection of numeric values, 132
Combination	A selection of items from a set where the order of selection is <i>not</i> important, 12
Commutative	a function whose order of parameters can be swapped without changing the final result, 29
Complement	the set of all the elements in the universe that do not appear in the original set, 4



Complexity Class	formula describing how the running time (or memory usage) of an algorithm changes relative to changes in the size (or quantity) of input, 225
Contradiction	a Boolean expression that is never true regardless of input values, 48
Control Structure	a technique which determines how or in what order instructions are processed, 199
Corner Case	occurs when system input is technically legal but unexpected, at the edge of an allowable range, or otherwise unusual, 209
Counter	a variable changed by a fixed amount at each iteration of a loop, 207

D

D Latch	one bit memory cell which holds the given data when the clock input is true, 192
Decision Table	a logical table expressing and analyzing conditions and actions for a certain problem domain, 62
Decoder	logic circuit that converts from input into one output representing the input's binary value, 188
DeMorgan's Law	a complement can be distributed into an expression if the union and intersections are flipped, 30
Disjoint	two sets that do not share any elements in common, 5

Disjunctive Normal Form a series of subexpressions all connected by OR operators. Each subexpression must consist of a series of distinct variables, each one possibly prefaced with a NOT, that are connected with ANDs. Also called Sum of Products., 45

Distributive a function that, applied to a parentheses group can be distributed into and applied individually to each element within that group, 29

Double Dabble a shift and add algorithm to translate binary into BCD, 109

Double Negation the opposite of the opposite of any expression is itself, 29

E

Edge Triggered Flip Flop a memory cell which acquires a value only at the moment the clock input changes, 193

Empty set the set containing no elements, 2

Encoder logic circuit that converts from one input line into an equivalent binary number, 187

End of File abbreviated EOF or EOS (end of stream), a Boolean condition that indicates when no more values remain to be read from a list, 212

Equivalent two sets which contain the same elements are equivalent, 3

Error Correcting Code a technique for encoding a sequence of bits such that certain kinds of transmission errors can be detected, and in some cases, automatically corrected, 155

Extended ASCII any 8-bit or more extension of ASCII encoding, 124



F

Factorial

the product of integers from 1 up thru some given number, indicated by the exclamation point, 10

Finite Set

a set containing a limited (although possibly very large) number of elements, 2

Flag

a Boolean variable which indicates if a certain condition has occurred, 215

Flag

a bit which indicates the presence or absence of a particular condition or setting, 147

Floating Point

representation of a number using a form of scientific notation, where the position of the decimal point is separated from the numeric digits, 112

Font

a set of symbols which represent various code points, 127

Full Adder

logic circuit that can add three bits, 173

G

GIF

Graphics Interchange Format, an 8-bit lossless compressed image format popular for small animations, 140

H

Half Adder

logic circuit that can add two bits, 171

Hamming Code

a simple error correcting code that can correct one error, 160

Hamming Distance

given two bit sequences of equal length, the number of positions in which the values differ, 156

Heuristic an algorithm that approximates a solution, usually in much less time than it would take to find an exact solution, 235

Hexadecimal base 16 number systems with digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 92

I

Idempotent a function that, given two equal values, returns that value as the result, 28

Identity a function that, given any parameter, returns that value as the result, 29

Implication an expression “ a implies b ” is true if, whenever a is true, b is guaranteed to be true. If a is false, b is irrelevant and the implication is automatically true, 41

Implicit Bit the leftmost one in binary normalized exponential form that is assumed to be present but not actually stored, 114

Indifferent Condition a condition whose value does not matter for certain rules, 65

Infinite Loop a loop whose body does not change the loop condition, causing the loop to repeat continuously, until the program is manually terminated, 205

Infinite Set a set containing an unlimited number of elements, usually defined mathematically, 2

Integer positive or negative whole number, 100

Intersection the set of all elements that appear in both of the original sets, 4

Invert in a binary sequence, replace all 1s with 0s, and all 0s with 1s, 103



Iteration a single run through the loop body, 205

L

Law of Excluded Middle

for any Boolean expression or value x , exactly one of x and $\neg x$ must be true, and one must be false, 54

Linear Search

checking each element in a sequence one at a time, from beginning to end, until the desired element is found or the end is reached, 225

Logic Circuit

Boolean operations, indicated with specific symbols, connected with wires which show order of operation, 73

Loop Body

the task(s) or structures run repeatedly while or until a condition is met, 205

M

Multiplicative Counting Rule

When selecting one item from each of several sets, the total number of possibilities is the product of the cardinalities of the sets, 10

N

Nested Loop

a loop within the body of another loop, 212

Number System

a notation for representing numbers using symbols, 90

O

Octal

base 8 number systems with digits 0 through 7, 92

Overflow

condition caused when the result of an arithmetic operation is too large for the number of bits available, 102

P

Palette

a selection of colors available for use in an image, 140

Parity Bit

a bit making the number of 1 bits in a message even or odd, as selected, 158

Partition Rule

A technique for counting the number of possible divisions of some set into various unequal groups, 17

Permutation

A selection of items from a set where the order of selection is important, 12

Pixel

the smallest component of an image that a device displays or prints, 132

Priority Encoder

an encoder which outputs the binary value equivalent to the highest active input line, 187

Proper Subset

all of the elements of a set are contained within another set, and the other set also has at least one additional element, 6

R

Raster

image data stored as a rectangular grid of colors, 131

Reuse

using a single block of code, expression, or part of a circuit for several purposes. Also referred to as “DRY”, short for “Don’t Repeat Yourself”, 83

S

Satisfiable

a Boolean expression that is true for at least one permutation of input values, 49

Set	an unordered collection of items without duplicates, 2
Set Difference	the set of all elements that appear in the first set but not the second set, 22
Short-Circuit Evaluation	portions of a Boolean expression may be skipped (unevaluated) if it is known that their value will not affect the final result, 40
Significand	portion of a floating point number consisting of the significant digits, 114
SR Latch	simple one bit memory cell which set either be Set (true) or Reset (false), 190
Subnormal Number	non-normalized numbers that fill in near zero to help avoid truncating to zero, 117
Subset	all of the elements of a set are contained within another set, 6
Symmetric Difference	the set of all elements that appear in the first set or the second set, but not both, 22

T

Tautology	a satisfiable Boolean expression that is always true, 49
Traveling Salesman Problem	an optimization problem which requests the ideal route between a series of locations, 234
Truth Table	a table indicating the true/false result of a Boolean expression for all possible permutations of input values, 37
Two's Complement	most common system for representing signed integers in binary; conversion between positive and negative is achieved by inverting and adding one, 103

U

UCS-2

Unary Operator

an early Unicode encoding which represented each code point with 16 bits, 125

Undefined Behavior

an operator that takes only one input, 39

actions may or may not occur when the conditions are not specified by a decision table, 70

Unidirectional Error

an error that can only occur in one direction; such as flipping a one to a zero, but not the other way around, 156

Union

the set of all elements that appear in either or both of the original sets, 4

Universal Logic Gate

a gate that can implement any other logic gate or circuit, 77

Universe

the set containing all possible items under consideration, 2

UTF-8

modern Unicode encoding which is backwards compatible with 7 bit ASCII and can represent code points up to U+10FFFF, 126

V

Vector

Venn Diagram

image data stored as a collection of shapes, 131

a visual representation of a set expression, 19