CIS 205 Study Guide
Fall 2012

Don Colton
Brigham Young University–Hawaii

November 22, 2012

This is a study guide for the CIS 205 class, Discrete Math 1, taught by Don Colton, Fall 2012, at Brigham Young University–Hawaii.

This study guide is focused directly on the grading of the course, as taught by Don Colton. Specifically, the exams are explained and the skills they measure are taught.

This study guide was created in August of 2012, with parts coming from previously written documents, and other parts being newly composed. The whole of it is a new document. There are probably errors, including typographical errors, grammatical errors, and possibly errors in the substance of what is taught. There are probably sections that could be worded better to be more understandable to students.

Extra credit is given to students that report errors of any kind, or improvements that can be made. See the chapter on grading for details.

Contents

Chapter 1

Semester Grade

Contents

Your semester grade for this course is calculated by the number of points you earn.

The exact details are stated in the syllabus, and further explained here. In case the syllabus disagrees with anything stated here, the syllabus shall prevail and be deemed correct.

I once read of a survey that was done among students. They were asked what portion of their semester grade should be based on effort, and what part should be based on demonstrated ability (performance). The average came out somewhere around 40% for effort and 60% for performance.

Although I am not convinced that the survey is reliable, still it does feel reasonable to me, so I have attempted in grading this class to divide the points between effort and performance using those guidelines.

We also provide some opportunities to earn extra credit.

1.1 Effort

Effort: About 40% of the points you can earn are for effort, even if you are unable to perform well after putting in the effort. Performance does not enter into it, although we hope your efforts will result in improved performance.

Using a 1000-point scale for the semester grade, about 400 points are devoted to effort.

Effort includes studying and in-class participation.

1.1.1 Study Time

For a brief summary, check the syllabus. Here we will try to clarify it by using more words. (Please excuse us for being so wordy.)

For study we ask you to certify that you studied a certain amount of time.

Over the course of 13 to 14 weeks, this three-credit class will meet about 40 hours, including time for the final exam itself.

It is generally expected in college-level education that students will study two to three hours outside of class for every hour they spend in class.

Minimum Standard: Our minimum standard will be somewhat less. We will expect 1.5 hours outside of class for each hour in class. This adds up to 7.5 hours of study time per typical week. This counts the three hours in class as study time, plus 4.5 additional hours out of class. (If you miss class, you are welcome to make up the study time in other ways.)

Extra Credit: For extra credit, we allow you to count an additional 1.5 hours per week, for a maximum of 9 hours of study per week. You can study more than this, and it may improve your skills, but it will not give you more points directly.

Weekly Reporting: Your study time is reported each week at the first class of the week. (In case you miss that opportunity to report, you are allowed to send an email as soon as you can with the same information.)

We assume you will probably record your time by computer, using a text-editor file, and you will simply cut and paste it each week when you report it. You can use other methods if you like.

Daily Recording: Your weekly report will consist of a listing of days in the week since the previous report. Normally this will be eight days, because it will start with the start of class on Monday last week, and it will end with the start of class on Monday this week.

For each day, you will report the total hours studied, and a brief sentence about how those hours were spent. Phrases like "attended class 1 hr" and "read text book 2 hrs" would be sufficient.

The following activities qualify for study points:

Attend Class: Attending class and paying attention counts as study time. Time spent reviewing your notes from class also counts. (Time spent sleeping in class, doing email, updating Facebook, or watching sports or cat videos would not qualify, for example.)

Text Book: Reading and pondering the text book counts as study time. It can be any part of the book. You are not limited to just things we are currently studying. It can include deep reading. It can include skimming through the pages looking for something interesting.

Study Guide: Reading and pondering materials that I provide counts as study time. I mean things such as this study guide, or the syllabus, or emails I send relating to this class.

Study Group: If you become part of a study group, the time spent studying together for this class counts as study time.

Tutoring: If you seek tutoring (or provide it), the time spent in tutoring for this class counts as study time.

Other: If you think of another category that should qualify as study time, check with me so I can decide whether to approve it or not.

Interruptions: Minor interruptions, like a short phone call, do not count against you. Major interruptions should stop the clock.

Contemporaneous Record: To receive points for your study time, you must make a contemporaneous record of the time you spend studying. Contemporaneous means "at the same (con) time (temp)", by which we mean that you will keep a log in which you record the amount of time spent studying, and that you update it within 24 hours of when the studying actually occurred. So, Tuesday morning when you wake up, you could record, from memory, the time you spent studying on Monday. But Monday morning before class you would **NOT** record from memory all the time you spent during the previous week. That would not be contemporaneous enough for me.

Study hours will be multiplied by 4 to give points toward the semester grade. The normal expectation of 7.5 hours yields 30 points. The extreme possibility of 9.0 hours yields 36 points (including 30 points of regular credit and 6 points of extra credit).

Over the course of the semester, we have about 400 points plus up to about 80 points of extra credit.

1.1.2 Participation

For in-class participation, I don't know what I will measure exactly. I am still thinking about it.

1.2 Performance

Performance: About 60% of the points you can earn are for performance, even if it just comes naturally to you with no effort.

Using a 1000-point scale for the semester grade, about 600 points are devoted to performance.

Performance includes correctly answering questions on exams.

It also includes writing a few programs as assigned.

The remaining chapters of this study guide look at each of the performance points and give you the information we think you might need to master each skill.

1.2.1 Skills Exams

Skills Exams: For some exams, we will grade you based on the best performance you have during the semester. Each exam is made available at least twice, and possibly more times.

1.2.2 Other Exams

Other Exams: For some exams, grading is tedious. They may involve memorization of terminology. We try to limit the number of times we give such exams.

I have not yet decided the number of points to assign to each thing.

1.2.3 Programming

`http://gbot.dc.is2.byuh.edu/` is GradeBot Lite.

The list of available labs includes a number of labs with the number 205 at the start of the lab name.

Here is a preliminary list of the programs. The names may change, and programs may be added or deleted.

Programs:
- 205.p01.factors.all
- 205.p02.calc.fib - **Fibonacci**
- 205.p03.factors.perfect
- 205.p04.calc.gcd - greatest common divisor
- 205.p05.factors.prime
- 205.p11.mult - **multiplication rule**
- 205.p13.oswor - selections without replacement
- 205.p14.choose
- 205.p15.owii - orderings with identical items
- 205.p21.bst.BinSrchTree
- 205.p22.huffman - Huffman coding
- 205.p23.mst.MinSpanTree
- 205.p31.pq.PriorityQueue
- 205.p32.lcs.LngComSubseq
- 205.p33.zoo
- 205.p34.zoo.db
- 205.p35.nkrypto

See Chapter 3 (page 17) for more information on GradeBot and on what you need to know and do for these labs.

A number of these labs will be assigned throughout the semester. Your task will be to write the program in one of the supported languages (C, C++, Java, Perl, Python, or Tcl), probably Perl. When your program is working successfully, you will submit it for credit. (At that point it should be worth full credit.)

1.3 Extra Credit

Beyond the 40% for effort and the 60% for performance, extra credit is given for certain things.

Study time includes the opportunity for an extra 8% (80 points) toward your total grade.

Finding and reporting errors in this document is worth extra credit. If you **think** something is wrong or could be improved, let me know. I will decide if I agree that it is wrong or could be improved. If I agree, I will give you extra credit, typically 10 to 20 points, depending on my assessment of the thing you reported.

Other things may come up, where I want to encourage you to do something special, and I may offer extra credit for your doing of it. I don't know what those things will be, and there may not be any, but in the past it has happened a number of times.

Chapter 2

DCQuiz: My Learning Management System

Contents

I have developed my own learning management system (LMS) that will be used for this course. Other LMS examples include BlackBoard, Canvas, and Moodle. I did not write them. I currently do not use them.

`https://dcquiz.byuh.edu/` is the DCQuiz URL.

Since I wrote it myself, I am also responsible for any bugs that may be in its programming. If you notice any bugs, I hope you will let me know so I can get them fixed.

I can also make improvements when I think of them. I like that.

2.1 Grade Book

The most important place you will see DCQuiz is the grade book.

I use DCQuiz to manage my grade book for this class. You will be able to see the categories in which points are earned, and how many points are credited to you.

You will also be able to see how many points are credited to other students, but you will not be able to see which students they are.

This gives you the ability to see where you stand in the class, on a category-by-category basis, and in terms of overall points. Are you the top student? Are you the bottom? Are you comfortable with your standing?

2.2 Daily Update

Another place you are likely to see DCQuiz is the daily update.

Typically in class I start with a quiz called the Daily Update. It usually runs the first five minutes of class, and is followed by the opening prayer.

By having you log in and take the daily update quiz, I also get to see who is in class, in case I need a roll sheet and I did not take roll in some other way.

2.2.1 Calendar

Generally I show you the calendar of upcoming events. That way, if things change a bit from the syllabus, you are kept up to date.

2.2.2 Study Time

Generally I give you the opportunity to tell me how much study time you have accumulated since the last reporting. Study time is generally measured from start-of-class on Monday through start-of-class the next Monday. So, on Monday you would report the prior week. On Wednesday you would report the past 48 hours. One Friday you would add another 48 hours (making 96, or four days).

This lets me update your study points.

2.2.3 Comment

Generally I also give you an opportunity to make an anonymous comment. This can be anything you want to say. It might include announcements, such as birthdays or concerts. It might include questions. It can be a simple greeting.

Comments provide a chance for each student to say something without the embarrassment of everyone else knowing who said it. You can say how unfair you think I am for something. You can ask about something you find confusing.

I introduce it like this:

If you wish, you can type in a comment, question, announcement, or other statement at the start of class for us to consider. Or you can leave this blank.

This is a good opportunity to ask about something you find confusing.

The identity of the questioner (you) will not be disclosed to the class, and normally I will not check (although I could). My goal is for this to be anonymous.

2.2.4 Genuine Questions

I may include genuine questions in the daily update, and these can be graded. It's kind of unpredictable.

2.3 Exams

DCQuiz was originally developed for giving tests. My problem was handwriting, actually. Students would take tests on paper and sometimes I could not read what they had written. Sad, huh?

So I cobbled together an early version of DCQuiz to present the questions and collect the answers.

I got a couple of additional wonderful benefits, almost immediately.

First, I got the ability to grade students anonymously. All I was seeing was their answer. Not their handwriting. Not the color of their ink. Not their name at the top of the paper. It was wonderful. I could grade things without so much worry about whether every student was being treated fairly.

Second, I got the ability to share my grading results with every student in the class. Each student can see, not only the scores earned by other students, but the actual answers that other students put to each question. This gives students the ability to learn from each other.

Third, it gave students a way to verify that they were being graded fairly compared to their fellow students. If you can see your own answer, and see that everyone with higher points gave a better answer, that is a good thing. If you think your answer is better, it gives you a reason to come and see the teacher so you can argue for more points, or you can be taught the reasons for their answers getting more points.

Fourth, it gave me the convenience of grading anywhere without carrying a stack of papers. I could grade on vacation. (Wait. Doesn't that make it a not-vacation?) I could grade in class, or in my office, or at home.

Fifth, although I never did this, it theoretically has the ability for me to let other people be graders. But I never did this.

2.3.1 Taking Exams

As it currently operates, DCQuiz lets you, the student, log in and see a list of quizzes. (The grade book is actually just another quiz, but it is one where I enter grades that you earned some other way.)

Quizzes typically have starting and ending times. Before the quiz starts, there is a note telling when it will start. As the quiz gets closer, like within an hour or two, an actual count-down clock will appear telling you how long until the quiz is available.

Once you start the quiz, if it has an ending time, you will be able to see a count-down timer telling you how much time you have left.

As you take a quiz, you can see the main menu, the question menu, and the question page.

Main Menu: The main menu was already mentioned. That's where you see what quizzes are available.

Question Menu: The question menu shows you what questions are on this quiz. It lets you select a question to work on. It shows you which questions you have answered already. It shows you which answers have already been scored. It lets you say that you are done. It lets you cancel the quiz (if that is allowed).

Question Page: The question page shows a single question, and lets you type in your answer. Some questions only allow a single-line answer. When you press ENTER it takes you automatically to the next question. Other questions let you type in a number of lines.

Early Grading: The question page may allow an option for early grading. If you think you have given your final answer, you can submit it for early grading. If I have time, I will grade it while the test is still under way. That could give you confidence to answer related questions, knowing that you got something right.

Throwbacks: Along with early grading, I sometimes do something that I call a "throwback." That is when I look at your answer, and I think it is very close, but maybe you missed something. If so, I may unsubmit it for you. Then it will show up on your list of questions again. You can look at it and read the question again, and maybe realize what it was that you had not noticed before.

2.3.2 ezCalc

ezCalc is actually part of the Question Page, but it is special enough that I decided to let it have its own section in this study guide.

Many times a question will call for a numeric answer. In the early days of DCQuiz I had to make the questions easy enough to do in your head, or on paper, or else I had to give people access to a calculator. I was not comfortable letting them use a calculator because it might already be programmed for the kind of question I was asking.

Recently I found that too limiting, so I added a capability into DCQuiz to do simple calculations. I call it "ezCalc" and it can be used on single-line answers. Sometimes. If I decide to let it be used.

Say for instance that the answer to a question can be calculated as 7 x 6 x 5 x 4 x 3 x 2 x 1. (You probably recognize that as 7 factorial.) That may be easy to think about but hard to calculate. Beyond about 4 factorial, they are pretty hard to calculate without a calculator. But 4 factorial really limits the number of questions I can ask.

So I wrote ezCalc.

It works like this.

You key in a mathematical expression, like 7*6*5*4*3*2*1, and press the = key.

If ezCalc is available for that question, it will replace your expression with the answer, which is 5040 in this case.

ezCalc uses the "*" key for multiplication. It also has "+" for addition, "-" for subtraction, "/" for division, "%" for remainder (or modulus), and parentheses for grouping and controlling the order of operations.

To find out if ezCalc is available on a question, just type in something valid and simple like "1+2=". If it is available, your typing will be replaced by the answer, "3". If it is not available, your typing will not be replaced.

The purpose of ezCalc is to make it possible for me to ask harder questions but without letting big numbers get in your way. As long as you know how to do the calculation, ezCalc will do it for you.

ezCalc is not allowed to let you use variables or functions besides the ones I mentioned above. Sorry.

2.3.3 Reviewing Exams

When an exam is finished, DCQuiz lets me, the author of the exam, share it with you, the student who took the exam.

You can see reviewing opportunities on the main menu.

After selecting an exam to review, you will see a question menu similar to the one that was used for taking the exam. But instead of seeing your answer, you will see all the scores that were earned, with your score highlighted. If yours is the top score, it will appear first. If it is the bottom score, it will appear last.

You can select a question to drill down and see more details. Specifically, you can see each of the answers provided by each student that wrote an answer. And you can see the score it received. And you can see any notes the grader (me) may have made while grading.

This is intended to (a) let you teach yourself by seeing examples of work by other students, and (b) let you verify that you were graded fairly. (Every once in a while, maybe a few times per semester, a student will see that I entered their grade wrong, or I overlooked something. This is your chance to get errors fixed.)

Sometimes an exam is not open for review. The teacher gets to decide. But even if the exam is closed, you can still see the question menu (with the questions blanked out), and you can see your score and everyone else's score. Questions and answers are not available, but scores are available, even long after you took the test.

Sometimes an exam is deleted or revised and reused. The teacher gets to decide. When an exam is deleted, all questions and answers and scores are also deleted. After that, there is no way to see anything about that exam.

I generally revise and reuse the daily update exams. This causes all answers and scores to be deleted, but I keep the questions and just modify them for the next class meeting.

2.4 Homework

I generally use DCQuiz as the means for students to submit homework, if it is something that I want to look at carefully.

In cases like that, DCQuiz is generally set up to let you submit your homework from anywhere on the Internet, including from your dorm room or someplace on campus.

For exams, DCQuiz is generally set up to limit participation to those in the classroom, using classroom computers.

2.5 Other Features

DCQuiz has other features, such as the ability to limit where a test is taken, or to require a special code to access a test. Those features will be explained in class if they are ever needed.

Chapter 3

GBot: GradeBot Lite

Contents

3.1 Overview

`http://gbot.dc.is2.byuh.edu/` is the web interface for what used to be a huge system called GradeBot. What you see here is called GradeBot Lite.

GradeBot is an automated program grader. You write your program and upload it or paste it or key it into GradeBot. Then you press a button to have it graded. GradeBot will tell you if your program is successful or not.

The way GradeBot works is that it has a version of each program you are asked to write. It generates some sample inputs, runs its own version of the program, and collects the outputs. That is used to make a script.

You must follow the script. Your program is expected to behave exactly the same as GradeBot's program did.

This puts some serious constraints on your program. You must get all the strings right. If GradeBot wants "Please enter a number: " then that's exactly what your program must print. You may find yourself squinting at the output where GradeBot says you missed something. Usually it is a minor typographical error.

Once you get the first test right, GradeBot typically invents another test and has you run it. And another. And another. Eventually, you either make a mistake, or you get them all correct.

If you make a mistake, GradeBot will tell you what it was expecting, and what it got instead.

If you get everything correct, GradeBot will announce your success. That means your program's observable behavior is correct.

3.2 Submitting Your Work

When GradeBot thinks you have a working program, and you also think so, you can submit it for credit. It is totally okay to work ahead.

Submit it by emailing it to me at `doncolton2@gmail.com`

In the subject line of your email,

- Mention this class: CIS 205.
- Mention which program, by number, like p01.

In the body of your email,

• Tell me your name unless it is obvious from your email address.

• Specify which compiler should be used, unless it is obvious.

• Directly include the text of your program. (Do not use attachments. Do not use "rich text." Just use plain text.)

I will first of all cut and paste it into GBot and run it there myself. If it fails, you are not done, and I will refer it back to you to fix the errors.

Once it passes GBot, I will check the structure of your program. Is it constructed as required? And I will necessarily notice the style.

3.3 Interpreting GradeBot's Requirements

GradeBot provides input on the command line and by way of Standard Input. It expects output by way of Standard Output.

Quotes are shown in the examples to delimit the contents of the input and output lines. The quotes themselves are not present in the input, nor should they be placed in the output.

Each line ends with a newline character unless specified otherwise.

Standard Input

"`in>`" is shown to designate input that your program will be given (through the standard input channel).

Standard Output

Numbered lines are shown to designate output that your program must create.

"eof" indicates that your program must terminate cleanly.

Command Line Inputs

GradeBot will start your program by saying this:

```
GradeBot started your program with this command line:
```

It will then present the command line that was used to start your program. Often there is no special command line input, and the command line simply starts your program. Sometimes there are command line arguments. Here is an example:

```
‘‘./gcd 35 28’’
```

In this example, `./gcd` is the name of your program that is being run. `35` is the first command line argument. `28` is the second command line argument.

In most languages, command line arguments are available as an array named “argv” or something like it.

Return Codes

When your program terminates, it must send back a return code of zero unless something else was specified in the requirements. Many languages do this automatically.

For C programs, remember to start your program with “int main” and end it with “return 0;”

For other languages, do something similar if necessary.

3.4 Programming Style

From my point of view, style is really all about making your program easy to read and update. That is all I really care about.

Your code should be readable to a programmer that is unfamiliar with your programming language.

Use comments to explain what you are trying to accomplish with the key paragraphs of your program.

Indenting should correctly reveal the internal structure of your program. It should be consistent and reasonable. I personally like indenting by two spaces for each additional level of nesting.

Variable names should helpfully describe the contents they hold.

3.5 205.p01.factors.all

This is our "getting-started" task. It is a very simple program. If you can make it work, you should get full credit.

3.6 205.p02.calc.fib

(This task requires more background than I thought, so it is harder than it should be at this point in the lineup.)

The Fibonacci assignment builds (rapidly) on your introductory abilities by introducing two concepts: recursion and memo-ization.

The definition of the Fibonacci number series is this:

- `fib(1)=1` (basis case)
- `fib(2)=1` (basis case)
- `fib(n)=fib(n-1)+fib(n-2)` for higher numbers

For this class we will require a recursive solution with memo-ization. You should have a subroutine named `fib` and a static private array named `_fib` with size 46 or larger. If your language does not support that, you can use a global variable `_fib` instead. (`_fib` is your memo pad.)

Your main program should do all the input and output. It should have one call to the `fib` subroutine to calculate the result. Your fib subroutine should check to see if the requested value has already been calculated, and if so, simply return it. If not, it should make recursive calls to itself as specified by GradeBot.

You may need some way to initialize the `_fib` array. In Perl, you can tell whether something is defined or not, and then do the proper initialization if it was not yet defined.

```
if ( defined $_fib[$n] ) { ... }
```

3.7 205.p03.factors.perfect

This is an exercise in finding factors and adding them up. We define a "perfect" number as one whose factors add up to the original number.

Example: 6

6 has the factors 1, 2, and 3. (And 6, but we don't count that one.)

When you add up 1 + 2 + 3, you get 6.

Therefore, 6 is perfect.

Another perfect number is 28.

28 has factors 1, 2, 4, 7, and 14. 1 + 2 + 4 + 7 + 14 = 28.

If the factors add up to less than the number, we say it is "deficient."

10 is deficient because 1 + 2 + 5 = 8, and 8 is less than 10.

If the factors add up to more than the number, we say it is "excessive."

12 is excessive because 1 + 2 + 3 + 4 + 6 = 16, and 16 is more than 12.

So, identify and add up the factors, and compare the tally with the original number. Easy.

3.8 205.p04.calc.gcd

Use Euclid's algorithm to calculate the Greatest Common Divisor of two numbers. See:

`http://en.wikipedia.org/wiki/Euclidean_algorithm#Concrete_example`

Essentially, Euclid's algorithm works as follows.

If we have two numbers x and y, the gcd(x,y) can be calculated as follows. If x and y are equal, then that value is the gcd.

If they are not equal, we can subtract the smaller from the larger. Say x is larger than y, then gcd(x,y) is equal to gcd(x-y,y), which is an easier problem.

Actually, using the mod function, we can replace x with x % y, effectively subtracting y as many times as possible, thus speeding up the whole process (assuming % can be calculated efficiently).

There is a nice recursive solution based on gcd(a,b) = gcd(b,a%b). It converges very quickly.

3.9 205.p05.factors.prime

Prime Factorization

You are given a positive integer (at STDIN). Report its prime factorization, smallest to largest. The product of a factorization is the original number. A prime factorization uses only prime numbers. A prime number has no exact divisors but itself and 1. All inputs are integers greater than 1.

For example, if you are given 12, you would print something like this (depending on what GradeBot tells you it wants):

```
The prime factor(s) of 12 are 2 2 3.
```

3.10 Terminology

As we describe some of the following tasks, we use words like **selection**, **universe**, **sample**, and **distinct**. We briefly explain those words here.

By **selection** we mean a choice of one item from among all possible items that are available. If the possible items are "a b c" then we have three possible choices. If we choose "b" then "b" is our selection.

By **universe** we mean the set (or multi-set) from which possible items can be selected.

By **sample** we mean some selection from among the items in the universe.

By **distinct** we mean different. Sometimes we have identical items. If we cannot tell them apart, and one cannot be distinguished from the other, then we consider them to be identical. In the list "a a a" if we select "a" we cannot tell whether it is the first, second, or third "a" that was selected. We have only one distinct choice. In the list "a b c" we have three distinct choices. In the list "a b a" we have two distinct choices.

By **identical** we mean the opposite of distinct.

By **ordered** we mean that the order matters: "a b c" is different than "b a c".

By **unordered** we mean that the order does not matter: "a b c" is the same as "b a c".

By **replacement** we mean that after an item is selected, it can be selected again. From among "a b c" if we select three times **with** replacement, we could get "a b b". If we select three times **without** replacement, we could not get "a b b" because the second "b" would not be available for selection again.

3.11 205.p11.mult

We count the number of distinct assignments possible.

The model is assigning paint colors to houses in a neighborhood. (Each house is painted all the same color.)

If you have four houses, and three colors, then there are 81 possible assignments.

There are 3 possible assignments for the first house.

There are 3 possible assignments for the second house.

There are 3 possible assignments for the third house.

There are 3 possible assignments for the fourth house.

3 * 3 * 3 * 3 = 81.

This is known as the multiplication rule.

3.12 205.p13.oswor

Ordered Selections WithOut Replacement

Given a universe (the number of distinct objects among which selection is made) and a sample (the number of objects selected), tell how many distinct results are possible. Order matters: a b c is not the same as b c a.

For 26 3, we have 26 choices for the first selection, 25 choices for the second, and 24 choices for the third.

So, the answer is 26 * 25 * 24 = 15600.

(26!) / (23!) = 15600.

3.13 205.p14.choose

Unordered Selections WithOut Replacement

Given a universe (the number of distinct objects among which selection is made) and a sample (the number of objects selected), tell how many distinct results are possible. Order does not matter: a b c is the same as b c a.

For 26 3, we have 26 choices for the first selection, 25 choices for the second, and 24 choices for the third. But there are six ways each resulting sample could have been drawn (3 * 2 * 1).

So, the answer is (26 * 25 * 24) / (3 * 2 * 1) = 2600.

(26!) / (23! * 3!) = 2600.

3.14 205.p15.owii

Orderings With Identical Items

Usage: "owii set1 set2 ..." where "set1" (etc) is the number of identical objects of type "1" among which selection is made.

For example, how many distinct ways can you arrange the letters a a b b c? This would be "owii 2 2 1" or 30.

Since there are five items, you have 5 * 4 * 3 * 2 * 1 ways to arrange them. But since two of those items are identical, you must divide by 2 * 1 to cancel out duplicates. And since two more are also identical, you must divide again.

(5 * 4 * 3 * 2 * 1) / (2 * 1 * 2 * 1 * 1) = 30.

(5!) / (2! * 2! * 1!) = 30.

3.15 205.p21.bst.BinSrchTree

Binary Search Trees are explained in Chapter 13 (page 76).

This task is an extra-credit opportunity.

See `http://en.wikipedia.org/wiki/Binary_search_tree`

See `http://en.wikipedia.org/wiki/Tree_traversal`

Your program must build a Binary Search Tree using the elements given to it. Then, on demand, it must traverse that tree and report the elements of the tree in the order requested.

Commands are read from Standard Input.

The first command to handle is “add n” where n is a number to be added to the binary search tree.

Another command to handle is “find n” which should result in “found” if that object is found in the tree, and “not found” otherwise.

Another command to handle is “preorder” which should result in a **preorder** traversal of the tree, reporting each item as it is visited.

Another command to handle is “inorder” which should result in an **in-order** traversal of the tree, reporting each item as it is visited.

Another command to handle is “postorder” which should result in a **postorder** traversal of the tree, reporting each item as it is visited.

Another command to handle is “flush”, which means to throw away the current binary search tree and start fresh with an empty tree.

Another command to handle is “quit” which should result in termination of the program.

3.16 205.p22.huffman

Huffman Coding is explained in Chapter 14 (page 80).

This task is an extra-credit opportunity.

See `http://en.wikipedia.org/wiki/Huffman_coding`

Your program must read lines from Standard Input, one by one, until the line “`#`” is read.

Each line is of the form “(letter) occurs (count) times”.

The required response is “Bits required for Huffman coding: (count)”.

You are not asked to reveal the Huffman code you used, but all correct Huffman codes will have the same bit count.

3.17 205.p23.mst.MinSpanTree

Minimum Spanning Trees are explained in Chapter 15 (page 84).

This task is an extra-credit opportunity.

See `http://en.wikipedia.org/wiki/Minimum_spanning_tree`

Your program must read lines from Standard Input, one by one, until the line "`##`" is read.

Each line represents an edge of a graph, and will be of the form "vertex1 vertex2 cost" where vertex1 and vertex2 are strings. Cost is an integer.

After all lines are read, you must discover a minimum spanning tree within the graph, and you must report its total cost.

3.18 Other Tasks

Other tasks, including descriptions, learning objectives, and hints, will be posted here as they are developed.

Chapter 4

Formal Logic

Contents

Formal Logic involves deductions that can be made from sets of statements.

Important words and concepts: and, or, not, xor, implies, equivalence, DeMorgan, tautology, contradiction, commutativity, associativity, wff (well-formed formula), truth tables, modus ponens, modus tollens.

4.1 Modus Ponens

Modus Ponens (MP) is the most famous example of Formal Logic. If you have two facts, A implies B, and A is True, then MP makes the deduction that B must be True.

For example, let's say we have two facts: (a) It is true that "I worked hard in this class" implies "I will receive a grade of A in this class," and (b) It is true that "I worked hard in this class."

We let A stand for "I worked hard in this class."

We let B stand for "I will receive a grade of A in this class."

We can restate our two facts as: (a) It is true that A implies B, and (b) A is true.

According to Modus Ponens, we can deduce that B is true. Specifically, "I will receive a grade of A in this class."

Modus Ponens is short for Modus Ponendo Ponens, which is Latin for "the way (modus) that affirms (ponendo) by affirming (ponens)"

`http://en.wikipedia.org/wiki/Modus_ponens` has more.

4.2 Modus Tollens

Modus Tollens (MT) is the other famous example. If you have two facts, A implies B, and B is False, then MT makes the deduction that A must be False.

Modus Tollens is short for Modus Tollendo Tollens, which is Latin for "the way (modus) that denies (tollendo) by denying (tollens)"

`http://en.wikipedia.org/wiki/Modus_tollens` has more.

4.3 Atomic Statements

The fundamental basis of Formal Logic is the statement. In Formal Logic, each statement is either True or False. In the real world, statements may not be so clear-cut. They may be ambiguous. They may be self-contradictory. Formal Logic describes a useful subset of the real world, but not the whole thing.

“This statement is False” is a classic example of a self-contradictory statement. It cannot be accurately represented in Formal Logic.

Atomic Statements are statements that **are not** composed of other statements. Atomic Variables are words or symbols each of which represent an Atomic Statement. The phrases “Atomic Statement”, “Atomic Variable”, and simply “Atomic” can be used interchangeably to mean the same thing.

Compound Statements are statements that **are** composed or constructed from other statements. This is done by using operators.

The truth value of a Compound Statement can be derived by looking at the truth values of the Atomic Statements of which it is constructed.

4.4 Basic Operators

Connecting the statements of Formal Logic, we have several well-known operators. The most important of these are And, Or, and Not.

The And and Or operators are **associative** and **commutative**, by which we mean for a given string of Ands, it does not matter what order they are in, or in what order they are evaluated. The same is true for Ors.

4.4.1 And

When two statements are combined by the word And, the resulting statement is True when both of the original statements are True. Otherwise, the resulting statement is False.

For any number of statements, And is presumed to be True unless one or more of the statements is False. Then And becomes False.

And is also called Conjunction.

And is often represented by the symbol $\wedge$, pronounced “and” or “wedge.”

We can express the meaning of And by showing a Truth Table. We list all possible assignments of True and False to A and B. There are four of them, as shown in the truth table.

A	B	$A \wedge B$
T	T	**T**
T	F	**F**
F	T	**F**
F	F	**F**

4.4.2 Or

When two statements are combined by the word Or, the resulting statement is True when either one of the original statements is True. When both of them are False, the resulting statement is False.

For any number of statements, Or is presumed to be False unless one or more of the statements is True. Then Or becomes True.

Or is also called Disjunction.

Or is often represented by the symbol $\vee$, pronounced "or" or "vee."

We can express the meaning of Or by showing a Truth Table.

A	B	$A \vee B$
T	T	**T**
T	F	**T**
F	T	**T**
F	F	**F**

Xor

The above version of Or is sometimes called and/or, meaning either or both.

There is another version of Or, called Xor, for Exclusive Or, meaning either but not both.

For any number of statements, Xor is True if the number of true statements is odd, and False if the number of true statements is even.

Xor is also called Parity.

We can express the meaning of Xor by showing a Truth Table.

A	B	A xor B
T	T	**F**
T	F	**T**
F	T	**T**
F	F	**F**

4.4.3 Not

For exactly one statement, Not changes the truth value from True to False, or from False to True. If A is True, then Not A is False. If A is False, then Not A is True. This is also called Dichotomy, which means exactly two options (True and False). As mentioned above, the Real World does not always bless us with Dichotomy, but it happens often enough to be useful.

Not is also called Negation.

Not is represented several different ways: A', $-A$, and $\bar{A}$.

We can express the meaning of Not by showing a Truth Table. Because only one variable is involved, we only need two rows.

A	$-A$
T	**F**
F	**T**

4.5 Additional Operators

There are other commonly-used operators. These include Implies and Equivalent.

4.5.1 Implies

When we say that A Implies B, we mean that any time A is True, B is also True. When A is False, we know nothing about B.

Implies is often represented by the symbol $\rightarrow$ (as in $A \rightarrow B$) or by the symbol $\Rightarrow$.

We can express the meaning of Implies by showing a Truth Table.

A	B	$A \to B$
T	T	**T**
T	F	**F**
F	T	**T**
F	F	**T**

People often find it confusing that when A is False, Implies is always True. But, by convention, that is the way Implies is defined.

The mere fact that A Implies B does not mean that A "causes" B. They may both be caused by some other thing, "C". Or maybe causation is not involved at all.

4.5.2 Equivalence

Equivalence: When two statements always have the same Truth value, we say that they are equivalent.

Equivalence is often represented by the symbol $\equiv$.

We can express the meaning of Equivalence by showing a Truth Table.

A	B	$A \equiv B$
T	T	**T**
T	F	**F**
F	T	**F**
F	F	**T**

4.6 Truth Tables and Proof

An equivalent way to write A Implies B is: Not A Or B.

$(A \to B) \equiv (-A \vee B)$

Let's prove it.

To verify that two statements are equivalent, we can construct a Truth Table listing all possible assignments of True and False to the atomic variables.

If there are N atomic statements, then there are 2^N possible assignments of True and False to those atomic statements.

A Truth Table will have 2^N rows, with each row representing one assignment of truth values to the atomic variables.

To prove $(A \to B) \equiv (-A \lor B)$, we can construct the following Truth Table:

A	B	$A \to B$	$-A$	$-A \lor B$
T	T	**T**	F	**T**
T	F	**F**	F	**F**
F	T	**T**	T	**T**
F	F	**T**	T	**T**

Since the third column $(A \to B)$ and the fifth column $(-A \lor B)$ have the same truth value (TFTT), the statements are equivalent.

4.7 DeMorgan's Laws

DeMorgan's laws deal with how a "not" outside of parentheses can be distributed across the pieces that are inside the parentheses.

Let's look at "not (A and B)".

DeMorgan changes this into "not A or not B".

Let's look at "not (A or B)".

DeMorgan changes this into "not A and not B".

DeMorgan's laws can easily be proved using truth tables.

Exam Question 1 (p.89)**:**
Use Truth Tables to Prove DeMorgan's Laws.

Acceptable Answer:
Construct the appropriate truth table.

4.8 Normal Forms

Taken as a whole, all the statements we know to be True are called our Knowledge Base, sometimes abbreviated KB.

Because there are equivalent ways of writing many things, things that look different from each other can actually still have the same ultimate truth value and meaning.

To grapple with that variability, several ways of writing things have been singled out as Normal Forms. Any KB can be restated in any of these Normal Forms.

Normal Forms are also called Canonical Forms.

It turns out that all Formal Logic operators (like Implies) can be restated using just the three main operators, And, Or, and Not.

4.8.1 CNF: Conjunctive Normal Form

CNF: Conjunctive Normal Form is when we state our knowledge as a series of conjunctions.

$A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G \wedge H ...$

Each part of the conjunction is a statement asserted to be true.

Within each of those parts, typically there are alternatives, at least one of which must be tree. Each part is therefore a disjunction.

$(A1 \vee A2 \vee A3) \wedge (B1 \vee B2) \wedge C \wedge ...$

Thus, CNF is a conjunction of disjunctions.

`http://en.wikipedia.org/wiki/Conjunctive_normal_form` has more.

4.8.2 DNF: Disjunctive Normal Form

DNF: Disjunctive Normal Form is when we state our knowledge as a series of disjunctions.

CNF is generally more useful, at least for our purposes.

Chapter 5

S1: Resolution

Contents

This chapter prepares you for the S1 (Skills 1) exam.

`http://quizgen.doncolton.com/` quiz 41 provides exam material for this skill.

Modus Ponens and Modus Tollens are classic rules of Formal Logic. There are many.

`http://en.wikipedia.org/wiki/History_of_logic` has a history of formal logic.

Fortunately, rather than memorize all those classic rules, there is a single method that always works in every case. It was invented by John Alan Robinson, a mathematician, in 1965 and is called Resolution.

http://en.wikipedia.org/wiki/Resolution_(logic) talks more about resolution.

http://en.wikipedia.org/wiki/J._Alan_Robinson tells about Dr. Robinson. As of 2012 he is still alive.

The resolution exam is introduced as follows.

Given a list of TRUE statements, simplify it as much as possible by using logic.

5.1 Basics

We will express our knowledge in CNF, conjunctive normal form.

The list of TRUE statements is called the knowledge base (the KB). We will add and delete statements to improve the list.

Each statement is called a (disjunctive) clause, and consists of an or-list of simple propositions. If it is in the KB, it is supposed to be TRUE.

Each simple proposition is either TRUE or FALSE. Put another way, for every proposition "p", the clause "p or not p" must be TRUE.

"I am dry" is a simple proposition, and "I have an umbrella" is a simple proposition. Typically we abbreviate propositions down to letters or short words for convenience. The statement "I am dry, or I don't have an umbrella" would be written as "dry or not umbrella" or "dry -umb" or even (d -u). Each such statement is called a clause. As we said, every clause in the KB is asserted to be TRUE.

5.2 CNF Notation

We will express our Knowledge Base like this:

(-a b) (a b) (a c)

By this we mean there are three clauses. The first is (-a b). The second is (a b). The third is (a c). They are implicitly joined together by $\wedge$, with this meaning:

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

Each of the clauses is meant as a disjunction. Thus, the KB has this meaning:

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

The minus signs mean Not.

5.3 Clause Simplification

We can drop any proposition we know to be FALSE from a clause because the whole clause must be TRUE, and the FALSE part clearly isn't helping. This is exactly like regular math where adding zero or multiplying by one does not change anything. If x+0=5 we can drop the +0 and be left with x=5. If (a b c) is TRUE, and we find out somehow that (-a) is TRUE, meaning that "a" is FALSE, then we can drop the "a" part, leaving (b c).

5.4 Reduction

Adding more propositions to a disjunctive clause (an or-list) cannot change it from TRUE to FALSE. That is the nature of OR. If we know (b) is TRUE, then (b c) is also TRUE, no matter what "c" is. As a direct result, if we have two clauses in the KB, and the little one is an exact subset of the big one, we can throw away the big one without losing any information. For example, if we know (I can drive), it does not help to also say (I can drive) or (I am rich). It provides no information about my riches.

5.5 A or Not A

If we have a clause that includes "(a) or (not a)" within it, we can delete it from the KB. The clause provides no information, since we already know that (a -a) is always TRUE.

5.6 Resolution

New clauses can be created from old clauses. This part is a little tricky, but it is very powerful. It is the heart of the Resolution method.

If we have the clause (a x), and we also have the clause (-a y), we can combine them to create a new clause. Notice that one includes (a) in its or-list, and the other includes the opposite, (-a).

By basic rule 1, “a” is either TRUE or FALSE. Here is the tricky part. IF “a” is TRUE, then “not a” must be FALSE, so the second clause simplifies into (y) is TRUE. On the other hand, if “a” is FALSE, then the first clause simplifies into (x) is TRUE.

We can splice the original clauses together to say (x y). More generally, if we have (lots of things) or “a”, and (other things) or “not a”, we can combine them into (lots of things) or (other things). That’s resolution.

5.7 Common Mistake

For resolution, we need exactly one statement that is true in one clause and false in the other clause.

Sometimes we have two such statements.

Say we have (a b c d e) (-a -b c d e).

It is tempting, but incorrect, to try to “splice” on (a b) and deduce (c d e).

The opposite of (a) is (-a), but the opposite of (a b) is not (-a -b).

The correct splicing would be as follows:

(a b c d e) (-a -b c d e) can be spliced along “a”.

The first part gives us (b c d e). The second part gives us (-b c d e). The combination gives us (-b b c d e).

Sadly, the (-b b) part makes this a useless statement.

(a b c d e) (-a -b c d e) can also be spliced along (b), but the result is similarly useless.

Bottom line: if there are two (or more) opposites, the result will not be helpful.

5.8 Procedure

To simplify the KB, we look at pairs of clauses. If one is a subset of the other, we delete the bigger one. If they invertedly share a proposition (one has "a" and the other has "-a") we use resolution to generate a new clause and insert it into the KB (unless the new clause was already there). We continue looking at all pairs of clauses until no more insertions or deletions can be made. Careful ordering can make the job faster. But whatever order you do things, you will always get the same result in the end.

5.9 Example

Given (-a b) (a b) (a c), resolve.

Solution: We know that (-a b) and (a b) resolve to (b).

Then, we know that (b) renders (anything b) useless.

Therefore, the solution is: (a c) (b).

5.10 Required Format for Grading

We call our required format **sorted CNF**.

All questions will be given to you in sorted CNF order.

Sorted CNF means conjunctive normal form where each clause is also internally sorted. The sorting order is - then [a-z]. Also the whole clauses are further sorted into alphabetical order.

For ease of grading, we require that your answers be given back to us in sorted CNF order.

We will verify that you have a single space between clauses. We will verify that you have a single space before and after each proposition.

As a result, every correct answer will look exactly like the answer we calculated, and can be graded automatically by exact match.

Chapter 6

S2: Big Oh Analysis

Contents

This chapter prepares you for the S2 (Skills 2) exam, which introduces the analysis of algorithms to determine their running time.

`http://quizgen.doncolton.com/` quiz 12 and 13 provide exam material for this skill.

Welcome to **A Quick Guide to Big Oh.**

6.1 What Are We Trying To Do?

The amount of collected data in the world gets bigger every day. Credit card transactions. Log files. Web hits. Customer lists. Bigger. More.

One programming challenge is to build systems that do not fall to their knees under the weight of higher speeds and bigger transaction counts. We want systems that are robust, systems that degrade gracefully rather than collapse and melt down.

“Big Oh” analysis is that branch of computer science that measures program performance by simply looking at the program itself. There is a lot one can tell by looking at the code. We cannot easily measure the actual speed in seconds, but we **can** tell whether doubling the input will double the running time, or whether things will be worse or better.

There are some algorithms that are more work to program, but give a faster program. Programmers must choose the best approach to their task, given what they know of the future loads their program must support.

6.2 Introduction

“Big Oh” is the popular name for running-time analysis of algorithms. It is generally acknowledged that although you can buy more memory or a faster CPU chip, these things will not save you if you are running an inefficient algorithm. Computer Science students learn this material (and more) in their introductory courses. It is helpful for IS students to also have a grasp of the basic terminology and to have the ability to measure (in Big Oh fashion) the running time of various programs.

The words “Big Oh” have reference to “on the Order of,” or “Order of magnitude.” Specifically it is applied to running times of programs. As the input grows, what happens to the running time of the program?

The phrase "Big Oh" itself is used loosely here. Precisely it means that the algorithm runs at least that fast. Theta (Θ) is a more precise term used in Computer Science, but we will use somewhat less precise but much more familiar terminology, making "tight big oh" technically equivalent to "theta."

6.3 n Times As Much Input

We want to know what happens to the running time of our program if we get n times as much input data. For each algorithm, just by looking at the program code, we can come to some reliable conclusions. We imagine n to be really large. Thousands. Millions. Billions.

6.3.1 Typical Algorithms

In this section we talk about some typical algorithms and tell what their big oh running time would be.

6.3.2 Constant-Time Algorithms

An example of a constant-time algorithm would be one to pick the first number from a list. It does not matter how long the list is. In one step we can pick the first number and then stop.

If we have n times as much input, the running time does not change. Or, it "changes" by a factor of one. When the running time does not change, we say the algorithm is $\Theta(1)$, "theta one," "big oh one," or constant.

A real-life example would be selecting a new employee by taking the first application on the pile. It would not matter how many applications were on the pile.

(On my Big-Oh quizzes, simple statements run in $\Theta(1)$ time. That is why they are called simple.)

Constant-time algorithms are the best possible algorithms, unless that time is very long.

6.3.3 Linear-Time Algorithms

Let's say we want to find the biggest number in a list, when the list is in unpredictable order. To find the biggest number, we must look at each entry in the list.

If we have n times as much input, the running time is n times longer. Such an algorithm is called $\Theta(n)$, "theta n," "order n," or "linear."

Linear algorithms are very common in IS programming, and are generally accepted as being efficient, unless there is a known way to do the job faster.

6.3.4 Logarithmic-Time Algorithms

Logarithmic algorithms have running times that grow more slowly than the size of the input. Double the input and the running time only gets a little longer. It does not double.

The classic example of a log-time algorithm is binary search. Take the (in)famous "guess my number" game. In this game, I think of a number and you must guess it. On each turn, you make a guess and I tell you whether you are too high, too low, or just right. If my number is between 1 and 100, your first guess may be 50. By guessing 50, you cut in half the number of remaining possibilities. Say my number is 78, but you don't know that. You say 50. I say higher. You say 75. I say higher. You say 88. I say lower. You say 81. I say lower. Each step you narrow the possibilities by half (roughly).

In this game, if we were to double the initial range, making it between 1 and 200, would it take you twice as many guesses? No. One extra guess at the front would determine whether it was above or below 100. From there, we are back to the same original challenge.

If we have n times as much input (meaning n times as many numbers to search), it will take us $\log_2 n$ steps before we get down to the original input size. Algorithms that run in log time are said to have a running time of $\Theta(\lg n)$, "theta log n," "big oh log n," or simply "log n."

6.3.5 Root-n-Time Algorithms

Root-n algorithms run in time proportional to the square root of n (the input size). An example would be finding whether a number n is prime. To be prime, a number must not be the mathematical result of multiplying too smaller numbers. To find if a number is prime, we can test all the smaller numbers to see if they divide exactly into n. But there is a trick. If n is 101, we can stop when we have tested 10, because if 11 goes in, then 101 = 11 * a, and a must be smaller than 11. But since we have tested all the numbers smaller than 11, we can quit. Without even trying it, we know 11 could not work. Such an algorithm would have running time $\Theta(\sqrt{n})$, or "root n."

6.3.6 Exponential-Time Algorithms

If you are just preparing for the quiz, you can skip this section. There are no exponential-time algorithms on the quiz.

Just as logarithmic algorithms are not much affected by a doubling of the input, there are other algorithms that may work well up to a point, but then the running time seems to explode.

The classical example of an exponential algorithm is the "Traveling Salesman Problem" (TSP). This problem is much studied in theoretical computer science. The task is simple. Given n cities, a traveling salesman must visit each exactly once before returning home. The goal is to do it the fastest possible way (or cheapest or shortest). Under the most general assumptions, the only way known to reliably solve the problem is to look at every possible route and then pick the best one. There is no known way to eliminate a meaningful proportion of the routes without checking each one.

How many routes are there? n factorial. That is, n possibilities for the first visit, and $n - 1$ for the second visit, until eventually there is just one city left for the last visit.

We like to stay away from algorithms that are exponential. Instead we invent "heuristics" which are shortcuts that tend to give good results but are not guaranteed to be the absolute best. A heuristic for TSP might be: go next to the nearest unvisited city. Or, link up the closest pair of cities. Then link the next closest pair of cities. Good heuristics can be rather tricky, but the payoff is a programming solution that you can use before the salesman dies of old age.

We say that exponential algorithms run in $\Theta(e^x)$ or exponential time. There are substantial differences between exponential algorithms, but we will leave that discussion for the CS students.

6.4 Loops

The running time of a simple loop (nothing but simple statements inside it) depends on how many times the loop will execute. We will look at several simple cases.

6.4.1 Counting Up to n

The most common case is a loop whose index starts at one (or zero) and counts by ones up to some limit n. This is a $\Theta(n)$ loop, the most common type of loop.

6.4.2 Counting Down from n

Another common case is a loop whose index starts at n and counts by ones down to some set limit, usually one or zero. This is also a $\Theta(n)$ loop, (still) the most common type of loop.

6.4.3 Add/Subtract any Constant

Whether you count up (add) or down (subtract), and whether you count by ones or fives or tens, the result is still the same. Those factors do not affect the running time of the loop. It is still a $\Theta(n)$ loop.

6.4.4 Multiplying or Dividing

If you multiply by a constant greater than one, your running time will be $\Theta(\lg n)$. That is, your index starts at one, then doubles each time until you reach or exceed n. It does not matter whether you double each time, or multiply by three each time (or four or ten or one hundred). The running time is still log n.

Similarly, if you start at n and count down by dividing by two or three or ten at each step, stopping when you reach one (or ten or one hundred), the running time is also log n.

6.4.5 Unusual Limits

Watch especially for this one variation on the limit: $i * i < n$. In this case, we are running a loop where i starts at one, for instance, and steps up by a constant while $i * i < n$. This loop will not run the full n times, but will stop when i reaches $\sqrt{n}$. Thus, it becomes a root-n loop, written $\Theta(\sqrt{n})$.

If we step up or down by multiples, then the $i * i < n$ limit has no special effect. It would theoretically be $\Theta(\lg \sqrt{n})$, but mathematically this is still the same as $\Theta(\lg n)$.

6.5 Combinations of Algorithms

When we have a $\Theta(n)$ loop (block) buried inside another $\Theta(n)$ loop (block), the effects are multiplied. The total running time becomes $\Theta(n^2)$, or "n squared." An example would be comparing two unsorted lists to see if the same item is present in each list. We might take the first item from list one and compare it to each item in list two. That would time order n time. Then we repeat for the next item in list one. As we go through all n items in list one, we have $n \times n$ or n^2 comparisons. If we double the inputs, it takes us four times as long to complete the task.

6.5.1 Sequences of Statements

For a sequence of statements (including possibly whole blocks of statements), we take the worst running time among the statements.

For instance, a log n block followed by a linear block would have an overall running time that is linear. The effects of the log n loop just vanish. They are too small to worry about. There is an old saying in English: Take care of the dollars and the pennies will take care of themselves.

Simple statements run in $\Theta(1)$ time. That is why they are called simple. A series of however many simple statements still runs in $\Theta(1)$ time.

6.5.2 If-Else Constructs

For if-else constructs, we always assume the worst case when we are not sure what will happen. The worst case for an if-else construct is that it will do either the if side, or the else side, whichever one is worse. For practical purposes, this behaves the same as if we did both sides (see "sequences of statements" above).

6.5.3 Worst Running Time

In selecting the worst running time, we can follow two simple rules.

(1) If the running time includes a power of n, like n^2 or $n^{\frac{1}{2}}$ (which equals $\sqrt{n}$), then the block with the higher power of n is worse.

(2) If the powers of n are the same (or there are no powers of n), the block with more logs is worse.

For example, in comparing $n\sqrt{n}\lg n$ to $\lg^3 n$, the first has a power of n of 1.5 (one for n, a half for $\sqrt{n}$). The second has a power of n of zero. So the first is worse.

6.5.4 Nested Blocks

When the blocks (typically loops) are nested, we multiply their running times to get the overall running time.

6.5.5 Recursive Subroutines

If you are just preparing for the quiz, you can skip this section. There are no recursive subroutines on the quiz.

There is a more elaborate analysis that goes on for recursive subroutines. These are subroutines (or functions, or procedures) that call themselves. They are typical of a divide-and-conquer programming approach, where a function foo divides its input into smaller sets and calls itself, foo, on each of those sets.

This topic is covered in Computer Science courses.

6.6 And the Winner Is . . .

In the long run, a program with better running time is a better program. Some programs are not meant to live a long life. They run once or a few times and are permanently retired. For these programs, it does not matter much which algorithm you use (except exponential, which may not even finish once in your lifetime).

For any program that will run possibly many times, maybe for years and years, it is generally worth the extra effort to use the best possible algorithm. A simple algorithm is fast to program and takes longer to run. A more complex algorithm costs more to program, but then it runs faster forever.

It is important for every programmer to be able to do simple kinds of running-time analysis, such as those presented here. It is important to be able to identify a better algorithm by seeing that it has a faster running time.

Beyond that, for more information one should consult a basic book on Computer Science, or take a course in analysis of algorithms, where other aspects of Big Oh analysis are more fully explored.

Chapter 7

Sets

Contents

It is important for students of computing to know some things about sets. Particularly, it is important to know some terminology and how to perform a few operations.

A set is an unordered collection of things.

7.1 Finite Sets

The easiest thing (for me) to think about is a nice, small, finite set.

Let's take the days of the week. This is a set with seven items in it. Normally we consider it to be an ordered set, as follows:

Sun, Mon, Tue, Wed, Thu, Fri, Sat.

But we could do it in alphabetical order:

Fri, Mon, Sat, Sun, Thu, Tue, Wed.

It is still the same set. Order does not matter. (When order matters, we call it a list.)

We can define a set by listing its members inside curly braces {...}.

Days = { Sun, Mon, Tue, Wed, Thu, Fri, Sat }.

Days = { Fri, Mon, Sat, Sun, Thu, Tue, Wed }.

In both cases, the set has the same members. Order does not matter.

7.2 Operations on Sets

There are two really important operations on sets, but a few more that are also important. The really important ones are Union and Intersection. The others include Subtraction and Complement and Universe.

7.2.1 Membership

Say $X = \{ 1, 2, 3 \}$.

The elements of X are 1, 2, and 3.

We write $1 \in X$ to say that 1 is an element of X.

The symbol $\in$ is read "is an element of."

Notice that $\in$ looks like E in "element."

7.2.2 Union

The union of two (or more) sets is a set that includes all the members that are in any of the original sets.

Say $X = \{ 1, 2, 3 \}$ and $Y = \{ 3, 4, 5 \}$.

The union, written $X \cup Y$, is $\{ 1, 2, 3, 4, 5 \}$.

The symbol $\cup$ is read "union."

Notice that $\cup$ looks like U in "union."

Notice that X had three members, and Y had three members, and the union of X and Y had five members. The number 3 appears in each set. But duplicates count the same as singletons.

The number of members in a set is called its **cardinality**.

Exam Question 2 (p.89)**:**
What do we call the number of members in a set?

Acceptable Answer:
cardinality

Exam Question 3 (p.89)**:**
What does cardinality mean?

Acceptable Answer:
It is the number of members in a set.

7.2.3 Intersection

The intersection of two (or more) sets is a set that includes all the members that are in every one of the original sets.

Say $X = \{ 1, 2, 3 \}$ and $Y = \{ 3, 4, 5 \}$.

The intersection, written $X \cap Y$, is $\{ 3 \}$.

The symbol $\cap$ is read "intersection."

Notice that $\cap$ looks like an upside down $\cup$, and that intersections are kind of the opposite of unions.

7.2.4 Subtraction

The subtraction of one set from another is a set that includes all the members of the first set, except those that are also members of the second set.

Say $X = \{ 1, 2, 3 \}$ and $Y = \{ 3, 4, 5 \}$.

The subtraction, written $X - Y$, is $\{ 1, 2 \}$.

7.2.5 Universe

The universe, written U, is the set of all members of all sets in this category.

If we are talking about dinner foods, U would be all possible dinner foods.

7.2.6 Complement

The complement of a set X, written X', is just $U - X$, everything from the universe that is not in the original set.

7.2.7 Empty Set

The empty set is a special set defined as $\{ \}$, or the set with no members. It is also represented as $\varnothing$ and as $\emptyset$.

7.2.8 Disjoint Sets

When the intersection of two sets, X and Y, is the empty set $\varnothing$, then they have noting in common, and we say that X and Y are disjoint.

7.2.9 Subset

For any sets X and Y, if $X - Y = \varnothing$, it means that starting with X, and then removing everything that is also in Y, we are left with nothing. X is then recognized as a subset of Y.

The union of X and Y is Y, because X adds nothing new to Y.

The intersection of X and Y is X, because everything in X is also in Y.

$\varnothing$ is a subset of every set.

7.2.10 Proper Subset

If X is not equal to Y, then we say that X is a proper subset of Y.

$\varnothing$ is a proper subset of every set except itself.

7.2.11 Power Set

The power set of a set X is the set of all subsets of X.

For example, if $X = \{ 1, 2, 3 \}$, then the power set of X is { { }, { 1 }, { 2 }, { 3 }, { 1, 2 }, { 1, 3 }, { 2, 3 }, { 1, 2, 3 } },

In this case, X has a cardinality of 3. The power set of X has a cardinality of 8. $8 = 2^3$. The cardinality of the power set is always 2 raised to the power of the cardinality of the original set. That is because for each member of the original set, it can be either present or absent in a given subset. Two choices, taken across the N items in the set.

7.2.12 Set Generators

We can generate sets by describing them. We don't have to list them explicitly. For example, we could say:

$X = \{x|x \in \mathbb{N}^1, x^2 < 20\}$

In this case, X consists of all elements, we will call each of them x, such that x is a member of the set $\mathbb{N}^1$, and x^2 is less than 20.

That leaves us with $X = \{1, 2, 3, 4\}$

7.3 Infinite Sets

Most sets I talk about are finite. The members can be listed, even if there are a lot of them. But they don't go on forever.

However, there are some really important sets that do go on forever.

The first of these is the set of natural numbers: 1, 2, 3, ...

http://en.wikipedia.org/wiki/Natural_numbers has more.

$\mathbb{N}$ is used as a symbol to represent the natural numbers, also called the counting numbers.

When we care, $\mathbb{N}^0$ is used as a symbol to represent the natural numbers with zero included.

When we care, $\mathbb{N}^1$ or $\mathbb{N}^*$ is used as a symbol to represent the natural numbers without zero included.

Together with zero and the negatives of the natural numbers, we have the set of integers.

`http://en.wikipedia.org/wiki/Integer` has more.

$\mathbb{Z}$ is used as a symbol to represent the integers.

7.3.1 Infinite Cardinality

Two sets have the same cardinality, which means they have the same number of members, if you can construct a one-to-one correspondence between them. That's the definition of same cardinality.

We can show that $\mathbb{N}^0$ and $\mathbb{N}^1$ have the same number of members. This is strange because clearly $\mathbb{N}^0$ includes zero as well as all the members of $\mathbb{N}^1$. So how can they be the same?

We merely have to construct a one-to-one correspondence, so that each and every member of $\mathbb{N}^0$ is paired with a unique member of $\mathbb{N}^1$, and each and every member of $\mathbb{N}^1$ is paired with a unique member of $\mathbb{N}^0$,

That correspondence is "add one". We can take any member of $\mathbb{N}^0$ and add one to it, giving us a member of $\mathbb{N}^1$. The correspondence is complete. Given any member of $\mathbb{N}^0$, we can tell you which member it matches in $\mathbb{N}^1$, and vice versa. The correspondence is complete. That is our proof, a proof by construction.

In other words, infinity plus one equals infinity.

We can also show that $\mathbb{N}$ has the same cardinality as the even number. It has the same cardinality as $\mathbb{Z}$. It has the same cardinality as $\mathbb{R}$, the rational numbers.

The cardinality of $\mathbb{N}$ is a countable infinity.

There is a bigger infinity, an uncountable infinity. The cardinality of the real numbers is one such infinity.

7.3.2 Infinite Set Generators

We can specify infinite sets by describing them. We can never list them explicitly. For example, we could say:

$X = \{x^2 | x \in \mathbb{N}^0\}$

In this case, X consists of all elements, we will call each of them x^2, such that x is an element of the set $\mathbb{N}^0$, the natural numbers starting with zero.

That leaves us with $X = \{0, 1, 4, 9, 16, ...\}$, the set of square numbers.

Chapter 8

S?: Set Exam

The Set exam, which has yet to be written, is planned to include the following tasks.

* Given two small finite sets X and Y whose members are listed explicitly, calculate their union $X \cup Y$.

* Given two small finite sets X and Y whose members are listed explicitly, calculate their intersection $X \cap Y$.

* Given two small finite sets X and Y whose members are listed explicitly, calculate their difference $X - Y$.

* Given a set constructor, like $X = \{x | x \in \mathbb{N}^1, x^2 < 20\}$, list the members of X explicitly.

tba: more to be added.

Chapter 9

S3: Counting

Contents

This chapter prepares you for the S3 (Skills 3) exam.

`http://quizgen.doncolton.com/` quiz 31 provides exam material for this skill.

We include here Combinations and Permutations.

9.1 What Are We Trying To Do?

It can be important to know approximately or exactly how many of a thing can exist, or how many ways a thing can be done.

In this chapter, we briefly look at six of those ways and show how to calculate the count for each.

9.2 Distinct Assignments

Say we have N houses and K colors of paint. Each house must be painted with one of those colors. (We keep it simple; there is no mixing of paint.)

How many ways can this be done?

The answer is that each house can be painted K different ways. Those ways do not influence each other in any way.

So, we have K ways for the first house, and K ways for the second house, and K ways for the third house, and so on.

We have K times K ..., for N Ks, or K^N, K to the Nth power.

This relies on the **multiplication rule**, where the total number of possibilities is calculated by multiplying the number of possibilities of each individual piece.

9.3 Permutation

Given N distinct (individual) objects, in how many different ways can they be listed (or arranged)?

The answer is $N!$, N factorial.

There are N choices for the item to be listed first. Then there are N-1 items from which we can choose the item to be listed second. Eventually, there is only one item that can be listed last.

$N!$ means $N \times (N-1) \times (N-2) \times (N-3) \times ... \times 3 \times 2 \times 1$.

The recursive definition is $N! = N \times (N-1)!$, and the basis case is $1! = 1$. (It is also generally agreed that $0! = 1$.)

Permutation is also called Ordered Full Set Without Replacement.

9.4 Ordered Selection

Short of a full permutation, we can stop after selecting K of the items. We then have an ordered selection of K items out of a total set of N items.

Just like with permutation, we calculate as follows:

There are N choices for the item to be listed first. Then there are N-1 items from which we can choose the item to be listed second. Eventually, there are N-K choices for the item that can be listed last.

$N \times (N-1) \times (N-2) \times (N-3) \times ... \times (N-K+1)$ which equals $N!/(N-K)!$.

If we want to list three items out of a set of seven possible items, the answer would be:

7! / 4! = 7 × 6 × 5 = 210

9.5 Unordered Selection

With an unordered selection, we don't care the order in which the items are selected. We can simply make an ordered selection, and then cancel out the different orderings in which the same items might appear.

This is also called "choose", as in "7 choose 3".

If we want an unordered group of three items out of a set of seven possible items, the answer would be:

7! / 4! = 7 × 6 × 5 = 210 for the ordered lists,

210 / 3 / 2 / 1 = 35 after noticing that there are 3 × 2 × 1 ways in which each of the selections can appear.

7! / 4! / 3! = 7 × 6 × 5 / 3 / 2 / 1 = 7 × 5 = 35

The formula is $N!/(N-K)!/K!$

9.6 Orderings With Identical Items

In the above examples, the items were always distinctive. They could always be told apart from one another.

When we have identical items, we assume the items cannot be told apart. Another way to say this is to call them "unlabeled" items.

A classic example is ordering letters to form "words."

In how many distinct ways can the letters *aaabbc* be arranged?

Well, if the letters were distinct, we would have 6!. But since they are not distinct, we have to divide by the amount of duplication that exists.

Since there are three “a”s in the string, we divide by 6. The “a”s could have been arranged as $a_1a_2a_3$ or $a_1a_3a_2$ or four other ways.

Since there are two “b”s in the string, we divide by 2. The “b”s could have been arranged as b_1b_2 or b_2b_1.

Since there is only one “c” we divide by 1 (i.e., make no adjustment). The “c”s could have been arranged as c_1 only.

The answer will then be:

6 × 5 × 4 × 3 × 2 × 1 / 3 / 2 / 1 / 2 / 1 / 1 = 60

That is, 6!/3!/2!/1! = 60

9.7 Distributing Objects Into Bins

In this case, we have N bins. Each bin is distinct and identifiable. Each could represent a child that is receiving gifts.

The objects may include duplicates. For example, we may want to distribute 2 baseballs among 3 children. How many ways can this be done?

There are six ways, listed as follows, with each digit telling how many baseballs a certain child got. (ABC means A for the first child, B for the second child, and C for the third child.)

200, 110, 101, 020, 011, 002

Nobody said it had to be fair.

The calculation for this is just about like Orderings With Identical Items. But not quite.

We introduce a fake item, the boundary line between bins (children). We will use X to represent the boundary, an b to represent the baseball.

200 can be written as bbXX.

110 can be written as bXbX.

002 can be written as XXbb.

In short, we can restate the problem as how many different words can be constructed from the letters “XXbb”.

That would be 4!/2!/2! = 4 × 3 × 2 × 1 / 2 / 1 / 2 / 1 = 4 × 3 / 2 = 6.

What if there are more items than just baseballs?

We use the **multiplication rule**, handling each of the items separately.

Say we have three children, two baseballs, and two bats.

We have 6 ways to divide the baseballs. We also have 6 ways to divide the bats. The total is 36.

Say we have 4 kids, 3 balls, 2 gloves, and 1 bat.

For the balls, we have XXXbbb = 6!/3!/3! = 20.

For the gloves, we have XXXgg = 5!/3!/2! = 10.

For the bat, we have XXXb = 4!/3!/1! = 4. Well, that's kind of obvious when you think about it. One bat, four kids? Four ways.

Now we multiply those together, since the assignments are independent:

20 × 10 × 4 = 800

There are 800 ways to distribute 3 balls, 2 gloves, and 1 bat among 4 kids.

Chapter 10

Discrete Probability

Discrete probability is based on a set of cases, each of which is equally likely to occur.

An example would be the rolling of a six-sided die. There is one chance in six that the die will end with a 4 showing on top. We say the probability is 1/6 or one in six.

The probability of an even number (2, 4, or 6) showing after the roll is 1/6 + 1/6 + 1/6 = 1/2.

To calculate the probability of a specific outcome, we need to count the number of equally-likely ways that outcome could happen, and the number of equally-likely ways **any** outcome could happen. Then we divide.

When rolling two six-sided **dice**, the probability of the total being seven is calculated as follows.

It could happen as (1 6) or (2 5) or (3 4) or (4 3) or (5 2) or (6 1). That makes six equally-likely ways it could happen.

There are 36 total equally-likely outcomes: (1 1), (1 2), (1 3), ..., (6 6).

It may seem that (1 6) and (6 1) are really the same, and in one sense that is true. But consider the **dice** to be of different colors, say yellow and red. There is only one way to roll (6 6). The yellow die must be 6 and the red die must be 6. But there are two ways to roll a one and a six. Yellow could be 1 and red 6. Or Yellow could be 6 and red 1.

We must be careful to measure equally-likely outcomes.

Example: What is the probability that a family with two children has one child of each gender (male / female)?

Wrong analysis: There are three possibilities: boy-boy, girl-girl, and one of each. Therefore the probability is 1/3.

Correct analysis: There are four equally-likely possibilities: boy-boy, boy-girl, girl-boy, and girl-girl. Therefore the probability is 2/4 or 1/2.

Independence

When two outcomes do not depend on each other, we call them independent.

Specifically, if we know one of the outcomes, and they are independent, then we do not know anything about the other outcome.

Consider Yellow/red, as used in the example above. After rolling the **dice**, if we know that yellow came up 4, we know nothing about red.

Yellow/red is independent.

On the other hand, knowing one outcome can give us information about the other outcome sometimes. If so, they are not independent.

Let's say the outcomes are (smallest number) and (biggest number) from the roll of a pair of **dice**.

If we know the smallest number is 1, then we have no information about the other die.

But if we know the smallest number is 4, then we can conclude that the biggest number is either 4, 5, or 6.

And if we know the smallest number is 6, then we can conclude that the biggest number is also 6.

Smallest/biggest is not independent.

Chapter 11

S4: Conditional Probability

Contents

This chapter prepares you for the S4 (Skills 4) exam.

`http://quizgen.doncolton.com/` quiz 45 provides exam material for this skill.

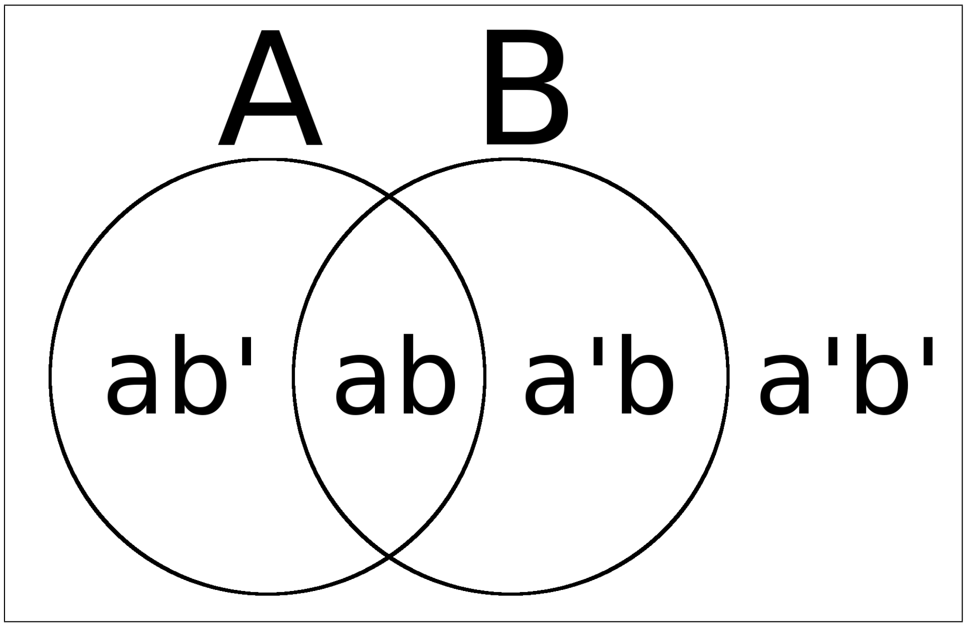
With conditional probability, we assume the outcomes or events are not independent. We assume that knowing something about one event gives us information about another event.

Bayes' Rule, discussed later in this chapter, is an important way of dealing with conditional probability.

First, we will look at conditional probability using Venn diagrams.

11.1 Strategy: Venn

Most students are familiar with the **Venn diagram**.



In this diagram, the circle on the left represents events in category A. The circle on the right represents events in category B.

The portion of circle A that is labeled ab' represents events that are in category A but not in category B.

The portion of circle B that is labeled a'b represents events that are in category B but not in category A.

The central area that is labeled ab represents events that are in both category A and category B. This is called the intersection of A and B.

The outside area that is labeled a'b' represents events that are neither in category A nor in category B.

http://en.wikipedia.org/wiki/Venn_diagram has a wonderful article that gives useful information. Even students familiar with Venn diagrams may learn something new.

The recommended approach to solving problems in conditional probability is to construct a Venn diagram using the facts at hand. Then, using the Venn diagram, determine the answer.

11.2 Notation

You should be familiar with the commonly used notations relating to probability. These are often written with mathematical symbols.

`http://en.wikipedia.org/wiki/Truth_table` has much more.

Primitives: These are some of the primitive wordings and operations used with probability.

$p(x)$: When you see p() read it as "the probability that ... is true".

$\cap$: When you see $\cap$ read it as the word "and". It can also be read as the word "intersection". It can be pronounced "cap". It is also called **conjunction**.

Notice that $\cap$ and $\wedge$ each represent the word "and". In the case of $\cap$ we are anding sets. In the case of $\wedge$ we are anding truth values. But they each mean "and."

$\cup$: When you see $\cup$ read it as the word "or". It can also be read as the word "union". It can be pronounced "cup". It is also called **disjunction**.

Notice that $\cup$ and $\vee$ each represent the word "or". In the case of $\cup$ we are oring sets. In the case of $\vee$ we are oring truth values. But they each mean "or."

$\overline{x}$: When you see a bar over something, read it as the word "not".

$|$: When you see $|$ read it as the word "given".

$\rightarrow$: When you see $\rightarrow$ read it as the word "implies". Note that "implies" is not the same as "causes".

Expressions: These are typical combinations of the five primitives into longer expressions.

$p(A)$ means "the probability that A is true".

$p(A)$=5/7 means that in the universe of possibilities, there are basically seven equally-likely groupings of things, and in five of them A is true.

$p(A{\cap}B)$ means the (joint) probability that both A and B are true. We may also write this as **p(A and B)**.

$p(\overline{B})$ means "the probability that not B is true", or in other words, "the probability that B is false". We may also write this as **p(not B)**.

$p(A{\cap}\overline{B})$ means the probability that A is true and B is false. We may also write this as **p(A and not B)**.

$p(A|B)$ means the probability that A is true if we already know that B is true. We may also write this as **p(A given B)**.

$p(A{\rightarrow}B)$ means the probability that if A is true, then B is also true. We may also write this as **p(A implies B)**.

11.3 Sample Problems

http://quizgen.doncolton.com/ quiz q45 provides additional opportunities for you to learn and practice these skills.

Given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

You should first derive the following Venn diagram: (4(2)5)7.

The "(4(2)" part represents the A circle in the **Venn diagram**. It means that in four cases, A is true but B is not true. In two cases A is true and B is true. It does not say anything about when A is false.

The "(2)5)" part represents the B circle in the Venn diagram. It means that in two cases, B is true and A is also true. In five cases B is true but A is not true. It does not say anything about when B is false.

The "7" part represents the space outside the A and B circles. It means that in seven cases both A and B are false.

Each of these 4, 2, 5, and 7 cases are independent and equally likely, for a total of 18 possible cases.

Exam Question 4 (p.89)**:**
Find $p(A \cap \overline{B})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

Acceptable Answer:
2/9

Exam Question 5 (p.89)**:**
Find $p(\overline{A}{\cap}B)$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

Acceptable Answer:
5/18

Exam Question 6 (p.89)**:**
Find $p(\overline{A} \cap \overline{B})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

Acceptable Answer:

7/18

Exam Question 7 (p.89)**:**
Find $p(A|B)$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

Acceptable Answer:
2/7

Exam Question 8 (p.89)**:**
Find $p(A|\overline{B})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

Acceptable Answer:
4/11

Exam Question 9 (p.89)**:**
Find $p(B|A)$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

Acceptable Answer:
1/3

Exam Question 10 (p.89)**:**
Find $p(B|\overline{A})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

Acceptable Answer:
5/12

Given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12, you should first derive the following Venn diagram: (3(7)1)1.

Exam Question 11 (p.89)**:**
Find $p(\overline{A}{\cap}B)$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

Acceptable Answer:
1/12

Exam Question 12 (p.89)**:**
Find $p(\overline{A} \cap \overline{B})$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

Acceptable Answer:
1/12

Exam Question 13 (p.89)**:**
Find $p(A|B)$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

Acceptable Answer:
7/8

Exam Question 14 (p.90)**:**
Find $p(A|\overline{B})$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

Acceptable Answer:
3/4

Exam Question 15 (p.90):
Find $p(B|A)$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

Acceptable Answer:
7/10

Exam Question 16 (p.90):
Find $p(B|\overline{A})$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

Acceptable Answer:
1/2

Given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21, you should first derive the following Venn diagram: (7(2)7)5.

Exam Question 17 (p.90):
Find $p(A \cap \overline{B})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

Acceptable Answer:
1/3

Exam Question 18 (p.90):
Find $p(\overline{A}{\cap}B)$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

Acceptable Answer:
1/3

Exam Question 19 (p.90):
Find $p(\overline{A} \cap \overline{B})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

Acceptable Answer:
5/21

Exam Question 20 (p.90):
Find $p(A|B)$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

Acceptable Answer:
2/9

Exam Question 21 (p.90):
Find $p(A|\overline{B})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

Acceptable Answer:
7/12

Exam Question 22 (p.90):

Find $p(B|A)$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

Acceptable Answer:
2/9

Exam Question 23 (p.90)**:**
Find $p(B|\overline{A})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

Acceptable Answer:
7/12

11.4 Bayesian Probability

http://en.wikipedia.org/wiki/Thomas_Bayes has information about Thomas Bayes (1702-1761).

Bayes comes up a lot in artificial intelligence. It is surprisingly simple.

11.4.1 The Product Rule

The **product rule** is this:

$p(a{\wedge}b) = p(a|b)p(b)$ or $p(a{\wedge}b) = p(b|a)p(a)$

This can be easily verified by looking at a Venn diagram.

We will show an example of this shortly.

Combining these two forms, we get **Bayes' rule**, aka Bayes' law, aka Bayes' theorem, which is this:

$p(b|a) = p(a|b)p(b)/p(a)$

(For some reason, I have a hard time remembering the Bayes' rule, but a somewhat easier time remembering the product rule. Fortunately for me, it is easy to derive Bayes from product.)

Each of $p(a)$ and $p(b)$ are called **prior probabilities**, or **a priori probabilities**. They are the probabilities that something is true when you don't know anything else about anything.

$p(a|b)$ is the conditional probability that a is true given that you already know b is true.

Exam Question 24 (p.90)**:**
What is the product rule?

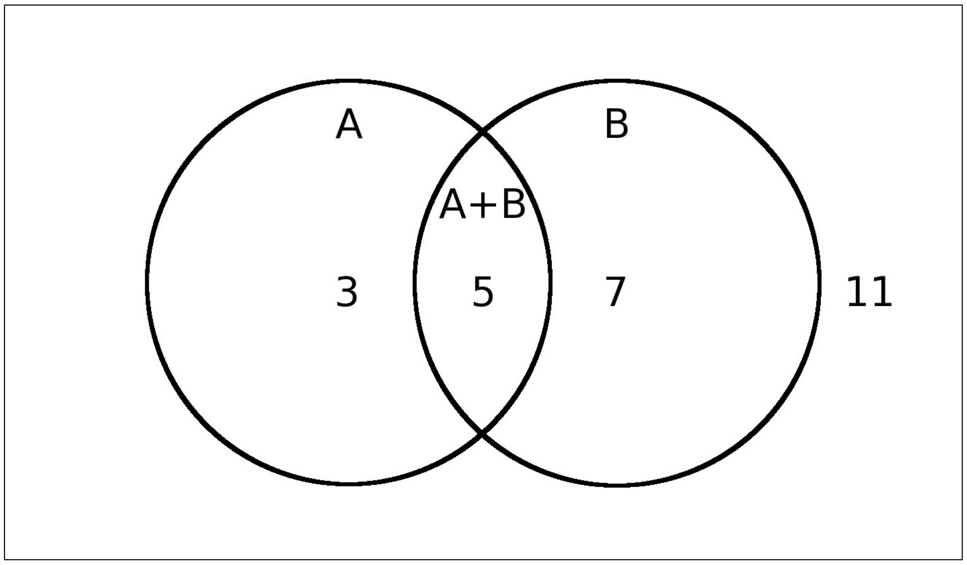
Acceptable Answer:
p(a and b) = p(a given b)p(b)

Exam Question 25 (p.90)**:**
What is Bayes' rule?

Acceptable Answer:
p(a given b) = p(b given a)p(a)/p(b)

11.4.2 Detailed Example

Let's look at what this means by way of a Venn diagram.



In this example, $p(A) = 8/26$, $p(A \wedge B) = 5/26$, and $p(B) = 12/26$.

Bayes' rule is used to calculate $p(b|a)$ when our basic facts include $p(a|b)$, $p(a)$, and $p(b)$. We do this as follows.

We know that $p(A \wedge B) = p(A|B)p(B)$. What does that mean? $p(A|B)$ is the probability of A given that B is known to be true.

In this case, we have 5+7=12 cases where B is true, and in 5 of them, A is also true. Hence, $p(A|B)$ is 5/12.

We now have our three facts: $p(A|B) = 5/12$, $p(A) = 8/26$, and $p(B) = 12/26$.

The product rule tells us that multiplying $p(A|B)$ by $p(B)$ we get $p(A \wedge B)$.

Specifically, 5/12 times 12/26 equals 5/26. Notice that the 12s cancel each other out.

The product rule tells us that dividing $p(A \wedge B)$ by $p(A)$ we get $p(B|A)$ which is our goal. 5/26 divided by 8/26 equals 5/8. Notice that the 26s cancel each other out.

We can check our work by counting up the chances directly. We see there are 8 chances for a to be true, and in 5 of them, b is also true. Therefore $p(b|a)$ is 5/8.

Bayes' rule just combines the two applications of the product rule.

$p(B|A) = p(A|B)p(B)/p(A)$

$p(B|A) = (5/12)(12/26)/(8/26)$

11.4.3 Bayes' Rule and Bayesian Updating

Let's look at two propositions.

A: The sound ah was uttered.

B: The computer thinks it heard the sound ah.

We want to know the probability that ah was uttered when the computer recognizes an ah: $p(u|r)$.

We can find the prior probability that ah was **uttered** by looking at a corpus of speech labeled by trained human transcribers. Let's make up a number and say that out of some corpus of speech, ah is uttered in one percent of the frames.

$p(u) = 0.01$

We can find the prior probability that ah was **recognized** by looking at a similar corpus of speech labeled by computer. Let's make up a number and say that in that corpus of speech, ah is recognized in two percent of the frames.

$p(r) = 0.02$

Due to confusion between similar phonemes, let's say that out of the times ah was actually uttered, it is recognized as the most likely phoneme 3/4 of the time.

$p(r|u) = 0.75$

From this we can calculate $p(u|r)$, the probability that ah was uttered given the computer recognized it.

$p(u|r) = p(r|u)p(u)/p(r) =$ 0.75 * 0.01 / 0.02 = 0.375.

I guess in this case our computer is not very accurate yet.

Chapter 12

Relations and Functions

tba: more to be added.

Chapter 13

S5: BST, Binary Search Trees

Contents

This chapter prepares you for the S5 (Skills 5) exam.

`http://quizgen.doncolton.com/` quiz 36 provides exam material for this skill.

Binary trees are important because they can organize data for quick look up of any item. They are easy to build and easy to use.

13.1 Tree Creation

Binary search trees are constructed by repeatedly inserting new numbers (or other sortable objects) into the tree, one by one. The objects in the tree are called nodes.

The first object becomes the root of the tree.

Each subsequent object is inserted into the tree by starting at the root. If it is an exact match, we stop. If it is less than the root, we follow the left branch of the tree. If it is greater than the root, we follow the right branch of the tree.

We repeat this process until we get an exact match or we run out of nodes.

If we get an exact match, the object is found and we are done. We do not modify the tree.

If we run out of nodes, the object is not found, and we create a new node at that location.

"Repeat this process" is called **recursion**.

With random inputs, the tree is typically pretty well balanced, by which we mean the number of steps needed to reach any node is on the order of the log (base 2) of the number of nodes in the whole tree.

Thus, with a million nodes in a well-balanced tree, we will only wander through 20 nodes before finding a match or knowing that there is no match.

13.2 Tree Search (Lookup)

The main purpose of a binary search tree is search. By this we mean that an element is sought within the tree. It is either found or not found.

The procedure for search is the same as the procedure for insertion, up until the point that the item is either found or not found.

During tree search, the tree does not change in any way.

If the tree is well-balanced, the search runs in $O(\lg n)$ time.

13.3 Tree Traversal

Starting at the root of the tree, we can visit every node. There are two main ways to do this. **Depth-First Search** (**DFS**) uses a **stack** and **recursion** and requires very little extra space to keep track of our progress. Breadth-First Search (BFS) uses a **queue** and requires much more space but can provide answers faster in some cases.

As we “visit” each node, we do whatever processing we had in mind. This could simply mean comparing the node to some standard, or printing some attribute of the node.

13.3.1 Pre-Order Traversal

With pre-order traversal, when we enter a node, we visit it immediately. Then we recursively go down the left-hand side of the sub tree. Then we recursively go down the right-hand side of the sub tree.

The effect is to zig-zag up and down the tree, tracing its outline, starting at the root and ending when all nodes have been visited.

13.3.2 In-Order Traversal

With in-order traversal, when we enter a node, first we recursively go down the left-hand side of the sub tree. Then we visit the node itself. Then we recursively go down the right-hand side of the sub tree.

The effect is to zig-zag up and down the tree, tracing its outline, starting at the root and ending when all nodes have been visited.

The in-order traversal will always result in a sorted list of nodes.

13.3.3 Post-Order Traversal

With post-order traversal, when we enter a node, first we recursively go down the left-hand side of the sub tree. Then we recursively go down the right-hand side of the sub tree. Lastly we visit the node itself.

The effect is to zig-zag up and down the tree, tracing its outline, starting at the root and ending when all nodes have been visited.

13.3.4 Breadth-First Traversal

With breadth-first traversal, we start with an empty queue which is our to-do list. Then we add the root node to that list. Then we repeat the following steps.

We remove the first node from the list and visit it. Then, if it exists, we add its left-hand node to the end of our to-do list. Then, if it exists, we add its right-hand node to the end of our to-do list.

The effect is to go from left to right across the tree, row by row, starting at the root and ending when all nodes have been visited.

13.4 The Exam

You will be given a list of numbers to insert into a new binary search tree. It typically takes about two minutes to draw a 15-node tree on a sheet of paper.

You will then be asked to traverse the tree and report the numbers as they are visited. It typically takes 30 seconds to traverse a 15-node tree, typing the numbers into the computer as you go.

For efficiency, you may be asked to traverse the same tree several times in different ways.

Chapter 14

S6: Huffman Coding

Contents

This chapter prepares you for the S6 (Skills 6) exam.

http://quizgen.doncolton.com/ quiz 35 provides exam material for this skill.

14.1 Background

Huffman codes are also called prefix codes. Each letter (or other thing) is coded with a unique set of bits, such that no set of bits is a prefix to any other set of bits.

The bits essentially define a path through a decision tree, where the answers are always at the end, not buried in the middle like they can be for binary search trees. When you reach the end, you copy out that letter. Then start over at the root of the tree with the next bit.

14.2 Building The Tree

When constructing a Huffman code, you must build a decision tree. The procedure for doing this is simple.

Consider each letter to be a terminal (ending) node in the decision tree. Each node has a weight that is the frequency of that letter.

Find the two nodes with the lowest weight. Create another node that joins them, and give it a weight that is the sum of the two original weights.

Example: If the two smallest weights are (12 a) and (16 b), create a new node called (28 a b). When you reach that node in the ultimate decision tree, you will know the letter will be either a or b.

If there are more than two nodes with the same lowest weight, it does not matter which two you pick to combine.

Continue this process, combining the two smallest nodes and creating a new node, until you have combined everything. Then you are done. You have your tree.

14.3 Building The Code

Throughout the tree, for each non-terminal node, assign 0 to one branch and 1 to the other branch. It does not matter which way you assign them.

Now, starting at the root, follow the path to each letter, one by one. Copy down the 1s and 0s along the path to that letter. The result is the Huffman code for that letter.

Because of the way the codes are assigned, it is impossible for any two letters to have the same code. And it is impossible for any letter's code to be a "prefix" of another letter's code.

Example: If 11100 is a code, then 1110 cannot be a code, and 111 cannot be a code, and 11 cannot be a code, and 1 cannot be a code.

The reason is simple. When you receive 11100, it can be decoded by following the decision tree, starting at the base (root), and then taking the 1 branch, and then the 1 branch, and then the 1 branch, and then the 0 branch, and then the 0 branch. That will end at the letter which was encoded.

If any of the prefixes were a letter code, it would mean the tree had a letter that was not on the outer edge (frontier) of the tree. But because of the way the tree was built, we know that could not happen.

14.4 The Exam

You will be given a list of letters and frequencies.

Your task is to develop a set of codes for the letters.

However, the exam does not (currently) ask you for those codes.

Then, you multiply each frequency by the number of bits in its code. That becomes the cost for coding that letter that many times. Add it up across all the letters and you have the total.

The exam asks you for the total.

Example:

Using the specified letter.frequency pairs, develop a correct Huffman code. Report the total bits used (sum of length times frequency for each letter). Count carefully.

a.3 b.24 c.2 d.16 e.16 f.21 g.4 h.2 j.15 k.17 m.26

This means the letter "a" appears 3 times, the letter "b" appears 24 times, and so forth.

From this you might decide on the following Huffman code:

a=111010 b=110 c=1110110 d=010 e=011 f=101
g=11100 h=1110111 j=1111 k=100 m=00

(There are lots of correct answers. This is just one of them.)

Since a happens 3 times and requires 6 bits each time, we count 3*6 bits.

Since b happens 24 times are requires 3 bits each time, we count 24*3 bits.

The calculator capability will allow you to type in a mathematical expression like this:

3*6+24*3+2*7+16*3+16*3+21*3+4*5+2*7+15*4+17*3+26*2=

When the = sign is pressed, the expression will be replaced with the answer, which is 460. There is only one correct total. All correct Huffman codes have the same total bits.

14.5 Only One Correct Answer?

As you process the letters, you combine branches in the decision tree. You must always combine to two smallest-weight branches to guarantee a correct solution. (Sometimes another combination will result in the same total, but it cannot be guaranteed.)

If there are several choices with the lowest weight (frequency), it does not matter which way you combine. The result is still provably optimal.

It is possible to end up with two letters that have the same frequency and yet end up with a different number of bits.

In one example I saw, (6 x) had five bits, and (6 y) had four bits. If you swap them around (which you might have), each x would take four bits, thus saving six bits, but each y would take five bits, thus costing six more bits. The net gain would be zero.

The order that things are combined affects the code you create, and maybe the number of bits per letter, but it does not affect the total bits needed.

Chapter 15

S7: MST, Minimum Spanning Trees

Contents

This chapter prepares you for the S7 (Skills 7) exam.

http://quizgen.doncolton.com/ quiz 18 provides exam material for this skill.

15.1 Background

A **graph** is a network of nodes, typically having many connections per node, and typically having **cycles** by which you can go from one node to another, eventually returning to your starting point.

Each connection between two nodes is called an **edge**.

A fully-connected graph that involves N nodes will have $N(N-1)/2$ edges.

A tree is a graph with no cycles, but where everything is connected. A tree that involves N nodes will have $N-1$ edges.

A **forest** is a graph with no cycles, but where everything may not be connected.

Sometimes the edges have weights (or costs).

A spanning tree is simply a tree that connects all the nodes of a graph.

A minimum spanning tree is a tree that connects all the nodes and has the smallest possible cost of any spanning tree.

We do not actually care how long any path is. It could be very long. The only thing we care about is that you can get from any point to any other point, and that the whole tree has minimal weight.

15.2 The Exam

You will be given a list of nodes (also called vertices). For example:

a b c d e f

In this example, there are six nodes and the first node is called “a”.

You will be given a list of edges in vertex.vertex.weight format. For example:

a.b.17 a.c.7 a.e.14 a.f.14 b.d.18 b.e.10 b.f.25 c.d.10 c.e.29 c.f.13 d.e.8 d.f.17

In this example, the first edge is between nodes a and b, and has a weight of 17.

Your task is to discover a minimum spanning tree and report its total weight.

You will be provided with a calculator capability to help you add up the weights, but you must decide which weights to add up.

The typical approach to solving this problem is to draw a diagram that shows each of the nodes, probably arranged in a circle. Then each edge is added by drawing a line between its two nodes, and its weight is written on or near the edge.

Next, you select a node to be your starting point. The choice does not matter because every node has to be part of the spanning tree eventually. The MST is grown by adding **the smallest edge that will add a new node to the tree.** Edges that would create cycles are crossed out. Eventually all edges will either be added or crossed out.

In the example above, the first node selected might be (d).

Next we might add d.e.8, thus adding (e) to the tree. We have to select an edge that involves d, and d.e.8 is the one with the lowest weight.

Next we might add b.e.10, thus adding (b) to the tree. We have to select an edge that involves d or e, and we have two choices that are minimal.

Next we might add c.d.10, thus adding (c) to the tree. We have to select an edge that involves d, e, or b, and something not already in the tree.

Next we might add a.c.7, thus adding (a) to the tree. We have to select an edge that involves d, e, b, or c, and something not already in the tree.

Next we might add c.f.13, thus adding (f) to the tree.

Finally, the weights of the included edges are added up and the answer is typed into the blank provided on the test.

The calculator capability will allow you to type in a mathematical expression like this:

8+10+10+7+13=

When the = sign is pressed, the expression will be replaced with the answer: 48

Chapter 16

Other Terms to Know

There are a number of questions you should be able to answer by the time you finish this class. You can find some of them in previous chapters of the study guide, and others right here. Those I am likely to test you on are listed in the Test Bank section, Appendix A (page 89).

tba: more to be added.

Exam Question 26 (p.90)**:**
possible question to be added

Acceptable Answer:
possible answer to be added

Unit I

Appendix

Appendix A

Test Bank

Test Bank

1: (p.34) Use Truth Tables to Prove DeMorgan's Laws.

2: (p.52) What do we call the number of members in a set?

3: (p.52) What does cardinality mean?

4: (p.68) Find $p(A \cap \overline{B})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

5: (p.68) Find $p(\overline{A}{\cap}B)$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

6: (p.68) Find $p(\overline{A} \cap \overline{B})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

7: (p.69) Find $p(A|B)$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

8: (p.69) Find $p(A|\overline{B})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

9: (p.69) Find $p(B|A)$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

10: (p.69) Find $p(B|\overline{A})$ given $p(A)$=1/3, $p(B)$=7/18, $p(A{\cap}B)$=1/9.

11: (p.69) Find $p(\overline{A}{\cap}B)$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

12: (p.69) Find $p(\overline{A} \cap \overline{B})$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

13: (p.69) Find $p(A|B)$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

14: (p.69) Find $p(A|\overline{B})$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

15: (p.70) Find $p(B|A)$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

16: (p.70) Find $p(B|\overline{A})$ given $p(A)$=5/6, $p(B)$=2/3, $p(A{\cap}B)$=7/12.

17: (p.70) Find $p(A \cap \overline{B})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

18: (p.70) Find $p(\overline{A}{\cap}B)$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

19: (p.70) Find $p(\overline{A} \cap \overline{B})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

20: (p.70) Find $p(A|B)$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

21: (p.70) Find $p(A|\overline{B})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

22: (p.70) Find $p(B|A)$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

23: (p.71) Find $p(B|\overline{A})$ given $p(A)$=3/7, $p(B)$=3/7, $p(A{\cap}B)$=2/21.

24: (p.71) What is the product rule?

25: (p.72) What is Bayes' rule?

26: (p.87) possible question to be added

Index