1 Administrivia and Syllabus

- **Time and place:** 2:15 - 3:35, Monday and Wednesday, in 102 Hall of Languages.
- **Required Texts:**
  - Data Structures and Other Objects Using Java, 3rd Ed or later. by M. Main, Addison-Wesley, 2005.
- **Suggested Texts:**
- **Course work:**
  - Your final grade will depend on:
    * Homeworks 20%
    * Labs 20%
    * Quizes/Tests 60%
Homeworks and Labs: There will be homeworaks every week or two and similarly with labs. People may work together in teams, but there are some restrictions. Homeworks and lab reports will not be accepted after their due dates. For the full details on homework assignments and labs follow this link.

- Quizes:
  - There will be a short quiz about every two weeks.
  - The Optional Final: The final will be a two-to-three hour, comprehensive test and is optional. If you take the final, your test grade will be the larger of: the cumulative grade on your in-class tests your grade on the final.

- Accommodations for a Disability: If you need an accommodation because of a disability, then contact both the instructor and the Office of Disability Services (Room 309 in 804 University Ave, phone: x 3-4498). Please do not be shy about this.

- Missed exams, labs, etc.: If you work or are involved in intercollegiate athletics, or similar, that requires your absence, contact me in advance. Arrangements are available.

- Honor Policy: Please see the separate handout. The form must be signed and returned before any homeworks, labs, or exams will be graded.

2 Monday, September 10, 2012

Homework 2.1:

- Read: On to Java (Winston & Narasimhan), Chapters 1 through 7 and 21 through 27.

- Do: Segments 126, 401, 436, 459, 469. In segment 126, the formula for the energy is \( \frac{1}{2} \times m \times v^2 \). In segment 401, for ideal body weight, use the Miller formula:
  
  - Formulas for Ideal Body Weight
  - For men 56.2 + 55.5 \((h - 1.524)\)
  - For women 53.1 + 53.5 \((h - 1.524)\)
  - \( w \) = weight in kilograms
  - \( h \) = height in meters

In segment 436, for a simple example of how to do interactive, command-line input, consider the following code suggested by Jim Royer:
```java
import java.io.*;
import java.util.Scanner;

public class Sample {
    public static void main(String argv[]) {
        // create a scanner for interactive input
        Scanner scin = new Scanner(System.in);
        int n;
        double x;

        // Ask for an int and print its square
        System.out.print("Please type in an int: ");
        n = scin.nextInt();
        System.out.println(" The square of the int is: " + n*n);

        // Ask for a double and print its square
        System.out.print("Please type in a double: ");
        x = scin.nextDouble();
        System.out.println(" The square of the double is: " + x*x);
    }
}
```

3 Wednesday, October 12, 2012

**Example 3.1**: Here is “blackboard” programming language code for a fast exponentiation.

```plaintext
{(N ≥ 0 and X > 0)}
K := N; B := X;
if odd(K) then A := X else A := 1;
K := K div 2;
while K > 0 do
    (B := B * B; if odd(K) then A := A * B;
    K := K div 2)
{A = X^N}
```

**Example 3.2** *(Fast matrix exponentiation)* The following code doesn’t work. Fix it for next week’s lab.
import java.io.*;
import java.util.Scanner;

public class MatrixFastExp {
    int[][] mult(int[][] X, int[][] Y) {
        int Z[][] = new int[2][2];
        Z[0][0] = X[0][0]*Y[0][0] + X[0][1]*Y[1][0];
        Z[0][1] = X[0][0]*Y[0][1] + X[0][1]*Y[1][1];
        Z[1][0] = X[1][0]*Y[0][0] + X[1][1]*Y[0][1];
        Z[1][1] = X[1][0]*Y[1][0] + X[1][1]*Y[1][1];
        return Z;
    }

    public static void main(String argv[]) {
        // create a scanner for interactive input
        Scanner scin = new Scanner(System.in);
        int matrix[][] = new int[2][2];
        int B[][] = new int[2][2];
        int A[][] = new int[2][2];
        int k,n;

        // Ask for the matrix
        System.out.print("[ M[0][0] M[0][1] ] = ");
        matrix[0][0] = scin.nextInt(); matrix[0][1] = scin.nextInt();
        System.out.print("[ M[1][0] M[1][1] ] = ");
        matrix[1][0] = scin.nextInt(); matrix[1][1] = scin.nextInt();
        System.out.print("n = ");
        n = scin.nextInt();

        k = n; B = matrix;
        if ((k%1) == 1) A = matrix;
        else {
            A[0][0] = 1; A[0][1] = 0;
            A[1][0] = 0; A[1][1] = 1;
        }
        while(k > 0) {
            B = mult(B,B);
            if ((k%1) == 1) A = mult(A,B);
            k = k/2;
        }
    }
}
System.out.println( A[0][0] + " " + A[0][1]);
System.out.println( A[1][0] + " " + A[1][1]);
}
}

4 Monday, October 24, 2012

Homework 4.1:

- **Read:**
  - *On to Java* (Winston & Narasimhan) Chapters 8 through 14.
  - *Data Structures and Other Objects Using Java* (Main) Chapters 1 through 3.

- **Do:**
  - *On to Java* (Winston & Narasimhan)
    1. Segment 188.
    3. Aggregate into a single exercise: segments 243, 244.
  - *Data Structures and Other Objects Using Java* (Main) Programming projects 8 and 9 on pages 97 and 98.
  - Implement classes for complex numbers with methods for adding, multiplying and conjugating complex numbers.

- **Note:** Provide suitable tests for each of your methods. ("Test code should demonstrate all of the methods of each class and all the features of each method.” - Jim Royer)

**Note 4.1:** Several of the following questions to be used as practice for the first exam are adapted from questions posed by Prof. Jim Royer in previous offerings of CIS 351.

**Question 4.1:** *(Practice Question 1 for 1st Exam)* Write a method

```java
public static int findMin(int a[])
```

that given `a`, an array of integers with `a.length` ≥ 1, returns the minimum value in this array. For example, if

```java
int tst1[] = {11,-6,29,-4}
```
then

\[ \text{System.out.println("The min of tst1 is " + findMin(tst1))} \]

would print:

The min of tst1 is -6

**Answer:** A full runnable class is given, but the `findMin` method alone is sufficient for the answer.

```java
public class MinInArray {
    public static void main(String argv[]) {
        int d[] = {11,-29,7,-5,101};
        System.out.println("The min of d[] is " + findMin(d));
    }

    public static int findMin(int a[]) {
        int minSoFar = a[0];
        for (int i = 1; i < a.length; i++)
            if (a[i] < minSoFar) minSoFar = a[i];
        return minSoFar;
    }
}
```

**Question 4.2:** (Practice Question 2 for 1st Exam) Write a method

```java
public static int findMinMaxSum(int a[])
```

that given a, an array of integers with \( a.length \geq 1 \), returns the sum of the minimum value and the maximum value in this array. For example, if

```java
int tst1[] = {11,-6,29,-4}
```

then

\[ \text{System.out.println("The minmaxsum of tst1 is " + findMinMaxSum(tst1))} \]

would print:

The minmaxsum of tst1 is 23

**Answer:** A full runnable class is given, but the `findMinMaxSum` method alone is sufficient for the answer. The important part of the answer is to see the use of `else` in the `findMinMaxSum` method: The boolean condition \((a[i] > maxSoFar)\) need not be evaluated if the `if` branch is executed.
public class MinMaxSumInArray {
    public static void main(String argv[]) {
        int d[] = {11,-29,7,-5,101};
        System.out.println("The minmaxsum of d[] is " + findMinMaxSum(d));
    }

    public static int findMinMaxSum(int a[]) {
        int minSoFar = a[0], maxSoFar = a[0];
        for (int i=1; i < a.length; i++)
            if (a[i] < minSoFar) minSoFar = a[i];
        else if (a[i] > maxSoFar) maxSoFar = a[i];
        return minSoFar + maxSoFar;
    }
}

Question 4.3: (Practice Question 3 for 1st Exam) Write a class Broadcast with a setter method that sets an integer variable N and whenever the setter sets this variable in any instance of Broadcast, the variable N gets that value in every instance of Broadcast.

Answer: Simply make the variable N and its setter method static.

Question 4.4: (Practice Question 4 for 1st Exam) The Ackermann-Peter function (let’s call it ack) maps a pair of nonnegative integers to an integer and is defined by

\[
\text{ack}(m,n) = \begin{cases} 
    n + 1, & \text{if } m = 0 \\
    \text{ack}(m-1,1), & \text{if } m > 0 \text{ and } n = 0 \\
    \text{ack}(m-1,\text{ack}(m,n-1)), & \text{if } m > 0 \text{ and } n > 0
\end{cases}
\]

Let

\[d(n) = \text{ack}(n,n)\]

Write a class that contains a method ack that implements the Ackermann-Peter function. Include in your ack method output commands that output the values of m and n whenever ack is recursively called, and output the value of ack(m,n) whenever ack(m,n) returns.

Beyond the scope of the practice question: Include a main that allows you to test your ack method. WARNING!!: The growth rate of \(d(n)\) as \(n\) increases is faster than the growth rate of almost any natural function you would ever implement. Do not try to compute \(d(5)\). The time complexity of the Ackermann-Peter function is even worse. Computing \(d(4)\) will significantly tax your resources. You may want to read about this function at Wikipedia, and you may want to poke around with google to find out about Rosza Péter.

Answer: The basic method without output to track the recursive calls is simply
public static int ack(int m, int n){
    return (m == 0) ? n+1 : ((n == 0) ? ack(m-1,1) : ack(m-1,ack(m,n-1)));}

Adding output to track the recursive calls might be done as follows:

public class Ackermann {

    public static int stackDepth;

    public static int ack(int m, int n){
        stackDepth += 1;
        printStackDepth(true,m,n);
        System.out.println();
        int a = (m == 0) ? n+1 : ((n == 0) ? ack(m-1,1) : ack(m-1,ack(m,n-1)));}
        stackDepth += -1;
        printStackDepth(false,m,n);
        System.out.println(a);
        return a;
    }

    public static void printStackDepth(boolean b, int m, int n){
        for(int i = 0; i < stackDepth; ++i) System.out.print("+");
        if (b) System.out.print(" Push: ack( + m + ", + n + ")");
        else System.out.print(" Pop: ack( + m + ", + n + ") = ");
    }

    public static void main(String argv[]){
        ack(2,2);
        //System.out.println("ack = "+ack(2,0));
    }
}

Here is the output corresponding to ack(2,2).

+ Push: ack(2,2)
++ Push: ack(2,1)
+++ Push: ack(2,0)
++++ Push: ack(1,1)
+++++ Push: ack(1,0)
++++++++ Push: ack(0,1)
+++++++++ Pop: ack(0,1) = 2
++++ Pop: ack(1,0) = 2
+++++ Push: ack(0,2)
++++ Pop: ack(0,2) = 3
+++ Pop: ack(1,1) = 3
++ Pop: ack(2,0) = 3


+++ Push: ack(1,3)
++++ Push: ack(1,2)
+++++ Push: ack(1,1)
++++++ Push: ack(1,0)
+++++++ Push: ack(0,1)
+++++++ Pop: ack(0,1) = 2
++++++ Pop: ack(1,0) = 2
+++++++ Push: ack(0,2)
++++++ Pop: ack(0,2) = 3
++++ Pop: ack(1,1) = 3
+++++++ Push: ack(0,3)
++++++ Pop: ack(0,3) = 4
+++ Pop: ack(1,2) = 4
++++ Push: ack(0,4)
+++ Pop: ack(0,4) = 5
++ Pop: ack(1,3) = 5
+ Pop: ack(2,1) = 5
++ Push: ack(1,5)
+++ Push: ack(1,4)
++++ Push: ack(1,3)
++++ Push: ack(1,2)
+++++ Push: ack(1,1)
++++++ Push: ack(1,0)
+++++++ Push: ack(0,1)
+++++++ Pop: ack(0,1) = 2
++++++ Pop: ack(1,0) = 2
+++++++ Push: ack(0,2)
++++++ Pop: ack(0,2) = 3
++++ Pop: ack(1,1) = 3
+++++++ Push: ack(0,3)
++++++ Pop: ack(0,3) = 4
++++ Pop: ack(1,2) = 4
++++ Push: ack(0,4)
++++ Pop: ack(0,4) = 5
+++ Pop: ack(1,3) = 5
++++ Push: ack(0,5)
+++ Pop: ack(0,5) = 6
++ Pop: ack(1,4) = 6
+++ Push: ack(0,6)
++ Pop: ack(0,6) = 7
+ Pop: ack(1,5) = 7
Pop: ack(2,2) = 7

△

Question 4.5: (Practice Question 5 for 1st Exam) Consider the following code:
public class ArrayTest{
    public static void main(String argv[]) {
        int a[] = {11}, b[] = {98}, c[] = {22};

        b[0] = a[0] + 4; c = b;
        System.out.println("First output: a[0] = " + a[0] + " b[0] = " +
                            b[0] + " c[0] = " + c[0]);

        c[0] = 9;
        System.out.println("Second output: a[0] = " + a[0] + " b[0] = " +
                            b[0] + " c[0] = " + c[0]);

        b = a;
        System.out.println("Third output: a[0] = " + a[0] + " b[0] = " +
                            b[0] + " c[0] = " + c[0]);
    }
}

What do the three output commands output? Explain. Hint: Compare the results to the following code. The results are different. Why?

public class IntVarTest{
    public static void main(String argv[]) {
        int a = 11, b = 98, c = 22;

        b = a + 4; c = b;
        System.out.println("First output: a = " + a + " b = " +
                            b + " c = " + c);

        c = 9;
        System.out.println("Second output: a = " + a + " b = " +
                            b + " c = " + c);

        b = a;
        System.out.println("Third output: a = " + a + " b = " +
                            b + " c = " + c);
    }
}

Answer:

In the ArrayTest class's main method when single cell integer arrays a, b and c are declared and initialized, memory is allocated for these cells. Array variable a has a hidden value which can be thought of as pointer to the array cell a[0], and similarly for array variables a and c. The three cells a[0], b[0], c[0] are initialized, respectively, to 11, 98 and 22. To
implement the command \( b[0] = a[0] + 4 \). The location of \( a[0] \) is accessed and the integer value stored at that location is returned, 4 is added to that value and the resulting integer value is stored in the location of \( b[0] \). The command \( c = b \) is implemented by changing the location of \( c[0] \) to become the location of \( b[0] \). Therefore, in the first output, the value of \( c[0] \) is the same as the value of \( b[0] \). The command \( c[0] = 9 \) changes the integer value in the location of \( c[0] \) to 9. But the location of \( b[0] \) is the same as the location of \( c[0] \), so the value of \( b[0] \) changes to 9 also. Then, the command \( b = a \) changes the location of \( b[0] \) to the location of \( a[0] \), but the location of \( c[0] \) remains unchanged.

In the case of the \texttt{IntVarTest} class, the locations of the variables \( a \), \( b \) and \( c \) never change, since primitive datatype values are stored in those locations, rather than hidden pointer values to locations where, ultimately, primitive values are stored.

\section{Dynamic 2D drawing}

\textbf{Example 5.1:}

```java
//file: NewHypno.java
import java.awt.*;
import java.awt.image.*;
//import java.awt.geom.GeneralPath;
import java.awt.geom.*;
import javax.swing.*;
import java.util.Random;
public class NewHypno extends JComponent implements Runnable {
    int imageWidth, imageHeight;
    double xLowerLeft, yLowerLeft, xUpperRight, yUpperRight;
    double pixelWidth, pixelHeight, invPixelWidth, invPixelHeight;
    double posX, posY;
    double[] pt = new double[2];
    double vertex0X, vertex0Y, vertex1X, vertex1Y, vertex2X, vertex2Y,
        vertex3X, vertex3Y, vertex4X, vertex4Y, vertex5X, vertex5Y,
        vertex6X, vertex6Y, vertex7X, vertex7Y;

    Random randNumGen = new Random(5);

    BufferedImage bufImage;
    Graphics2D bufImageGraphic;
```
public NewHypno(int w, int h, double xLL, double yLL, double xUR, double yUR)
{
    posX = 0.0; posY = 0.8;
    pt[0] = posX; pt[1] = posY;
    //vertex0X = 0; vertex0Y = vertex2Y = 0.1; vertex1X = 0.5;
    //vertex1Y = Math.sqrt(3)/2.0 + 0.1; vertex2X = 1;
    vertex0X = 0; vertex0Y = 0;
    vertex1X = 0; vertex1Y = 1;
    vertex2X = 1; vertex2Y = 0;
    vertex3X = 1; vertex3Y = 1;
    vertex4X = 0; vertex4Y = 0.5;
    vertex5X = 0.5; vertex5Y = 0;
    vertex6X = 1; vertex6Y = 0.5;
    vertex7X = 0.5; vertex7Y = 1;
    vertex3X = vertex3Y = vertex6X = vertex7Y = 1;
    vertex4Y = vertex5X = vertex6Y = vertex7X = 0.5;
    imageWidth = w; imageHeight = h;
    xLowerLeft = xLL; yLowerLeft = yLL;
    xUpperRight = xUR; yUpperRight = yUR;
    pixelWidth = (xUR - xLL)/((double)w);
    invPixelWidth = ((double)w)/(xUR - xLL);
    pixelHeight = (yUR - yLL)/((double)h);
    invPixelHeight = ((double)h)/(yUR - yLL);

    System.out.println(imageWidth + "    " + imageHeight);
    System.out.println(pixelWidth + "    " + pixelHeight);

    //Create an object with a buffer to hold an image w pixels wide
    //and h pixels high, using 8-bit RGB colors packed into integer
    //pixels:
    bufImage = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);

    //Create a graphics2D object to make the default
    //graphics2D methods available. The graphics object does not
    //know the size of the image. It knows only the origin and
    //orientation of the image's integer pixel coordinate system:
    bufImageGraphic = bufImage.createGraphics();

    //Prepare to draw a shape to be filled in with a color:
    bufImageGraphic.setPaint(Color.white); //white background version
    //bufImageGraphic.setPaint(Color.black); //black background version
//Fill in a w-by-h rectangular region with the previously
//selected paint color where the upper left vertex is located at pixel
//(0,0). These are "matrix coordinates". Since the image is also
//w-by-h and the upper left corners match, this rectangle will serve
//as the background of the image.
bufImageGraphic.fillRect(0, 0, w, h);

AffineTransform at =
    new AffineTransform(
        invPixelWidth, 0, 0, -invPixelHeight, -invPixelWidth*xLL,
        invPixelHeight*yUR
    );
bufImageGraphic.setTransform(at);

bufImageGraphic.setStroke(
    //new BasicStroke((float)pixelHeight)
    new BasicStroke(
        (float)(pixelWidth),
        BasicStroke.CAP_ROUND,
        BasicStroke.JOIN_ROUND
    )
);

Thread t = new Thread(this);
t.start();
}

public void run(){
    int count = 0;
    while (count < 5000000)
        try{
            timeStep(count);
            repaint();
            ++count;
            Thread.sleep(0);
        }
        catch (InterruptedException ie) {}
}

public Color makeColor(int n){
    Color col = (n == 0) ? Color.blue : Color.orange;
    return col;
}

public void timeStep(int n){
bufImageGraphic.setPaint(makeColor(n%2));

//Rectangle2D p =
//new Rectangle2D.Double(pt[0], pt[1], pixelWidth, pixelHeight);
Line2D p =
    new Line2D.Double(pt[0], pt[1], pt[0]+pixelWidth,
                        pt[1]+pixelHeight);
bufImageGraphic.draw(p);
pt = Sierpinski(pt);
}

public void paint(Graphics g){
    Graphics2D g2 = (Graphics2D)g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_OFF);
    g2.drawImage(bufImage, 0, 0, this);
}

public double[] Sierpinski(double[] pt)
{
    int target = Math.abs(randNumGen.nextInt( )) % 8;
    double[] q = new double[2];

    //If r = (1 - t)p + tq, then, as the value of t changes continuously from
    //0 to 1, r moves continuously from point p to point q. In the following
    //switch command, t = 2.0/3.0; so point pt jumps from it’s current
    //position to position 2.0/3.0 of the way to one of the vertices.
    switch (target){
    case 0 : q[0] = pt[0]/2.0 + vertex0X/2.0; q[1] = pt[1]/2.0 +
                        //vertex0Y/2.0; break;
    case 1 : q[0] = pt[0]/2.0 + vertex1X/2.0; q[1] = pt[1]/2.0 +
                        //vertex1Y/2.0; break;
    case 2 : q[0] = pt[0]/2.0 + vertex2X/2.0; q[1] = pt[1]/2.0 +
                        //vertex2Y/2.0; break;
    case 0 : q[0] = pt[0]/3.0 + (2.0*vertex0X)/3.0; q[1] = pt[1]/3.0 +
                        (2.0*vertex0Y)/3.0; break;
    case 1 : q[0] = pt[0]/3.0 + (2.0*vertex1X)/3.0; q[1] = pt[1]/3.0 +
                        (2.0*vertex1Y)/3.0; break;
    case 2 : q[0] = pt[0]/3.0 + (2.0*vertex2X)/3.0; q[1] = pt[1]/3.0 +
                        (2.0*vertex2Y)/3.0; break;
    case 3 : q[0] = pt[0]/3.0 + (2.0*vertex3X)/3.0; q[1] = pt[1]/3.0 +
                        (2.0*vertex3Y)/3.0; break;
    case 4 : q[0] = pt[0]/3.0 + (2.0*vertex4X)/3.0; q[1] = pt[1]/3.0 +
                        (2.0*vertex4Y)/3.0; break;
    case 5 : q[0] = pt[0]/3.0 + (2.0*vertex5X)/3.0; q[1] = pt[1]/3.0 +
                        (2.0*vertex5Y)/3.0; break;

public static void main(String[] args)
{
    JFrame frame = new JFrame("NewHypno");
    frame.setBackground(Color.white);
    NewHypno hypno = new NewHypno(700, 700, -0.1, -0.1, 1.1, 1.1);
    //NewHypno hypno = new NewHypno(700, 700, -2, -1.5, 1, 1.5);
    //NewHypno hypno = new NewHypno(700, 700, 0, 0, 700, 700);
    frame.getContentPane( ).add(hypno);
    frame.setSize(700, 700);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}

6 Big-O, also sometimes called asymptotics

(The material in this section is due almost in whole to Prof. James Royer. The material remains under his copyright and subject to those restrictions.)

6.1 Introduction

There are three basic kinds of data structures: primitive data types such as ints, doubles, chars; arrays; pointer structures. Then there are hybrid structures: a tree of "bed springs" of doubly linked lists of arrays of trees, for example. A variety of Insertion deletion and sorting algorithms is obvious. Mostly all anyone cares about is the time complexity of these operations. All key-comparison sorts have $n \log n$ lower bound complexity, relate that to the concept of information content, and tell you about how bubble sort gets around that in a phony kind of way. The $n \log n$ time complexity holds only because $n \log n$ and $\log n!$ are Theta of each other. The remaining topics: hashing and special kinds of trees. Time permitting, we will use Java to show how to use differential equations to draw animated
curvy things cheaply without knowing anything “calculus-wise” about differential equations. This is really just object oriented programming. But objects are considered data structures. The point is: data structures are to be used in software design to keep resource complexity, specifically time complexity, low. So without understanding time complexity what’s the point of data structures?

Consider a carpenter who is building you a porch or an addition to your house. You would not think much of this carpenter if he or she couldn’t produce a reasonable estimate of how much lumber this job will require — especially since this lumber is a major part of the bill you will be paying.

Now consider a programmer who is putting together a little application program for you. You would not think much of this programmer if he or she couldn’t tell you roughly how much time various parts of the program will take on different size inputs. The following covers the beginnings of how to make such estimates.

6.2 A beginning example

Let us start with looking a particular method that does something mildly interesting.

```java
public static int findMin(int a[]) {
    // PRECONDITION: a is not null.
    // RETURNS: the minimal value in
    // a[0], a[1], ..., a[a.length-1]
    int j, n, minVal;
    n = a.length; minVal = a[0];
    for (j=1; j < n; j++)
        if (a[j]<minVal) { minVal = a[j]; }
    return minVal;
}
```

Now let us ask our question

*How much time does this take?*

and notice that we are in trouble already.

Problem 1

Different arrays will produce different run times. For example, you would expect that an array of length 10000 will take longer to search than an array of length 3.

Problem 2

Different machines, different Java compilers and different Java runtime systems will generally produce different run times for `findMin`.

To deal with Problem 1, we can make our answer depend on the length of the array. That is, we can come up with a formula $F(n)$ such that $F(n)$ will be the runtime for an array of
length \( n \). Finding \( F \) exactly is usually hard, but it turns out that an estimate is almost always good enough.

To begin to deal with Problem 2, we make two assumptions about machines, compilers, and runtime systems.

**Assumption 1**

Given:

- a particular method \( M \),
- a particular setup, \( S_1 \), of computer, compiler, and runtime,
- another particular setup, \( S_2 \), of computer, compiler, and runtime,

then there are positive (real) constants \( c_{lower} \) and \( c_{upper} \) such that

\[
c_{lower} \leq \text{the } S_1 \text{ runtime of } M \text{ for an input of size } n \leq c_{upper}
\]

for sufficiently large values of \( n \).

Assumption 1 is just a careful way of saying things of the sort:

If we run the same program on the Woot\textsuperscript{TM} Mark V computer (which came out in 2001) and the Woot\textsuperscript{TM} Mark XI computer (which came out in 2008), we expect the Mark XI to be between 2.5 and 3.8 times faster than the Mark V — on reasonable size problems.

What does Assumption 1 have to do with Problem 2? It says that if we figure out an estimate for one setup, then we can turn this in to an estimate for another setup by finding (or estimating) \( c_{lower} \) and \( c_{upper} \).

**Example:** Suppose we know that \texttt{findMin} runs in \( F(n) \) microseconds on a Woot\textsuperscript{TM} Mark XI, then we know that on a Mark V \texttt{findMin} will run between \( 2.5 \cdot F(n) \) and \( 3.8 \cdot F(n) \) microseconds — on reasonable size arrays.

**Example:** Suppose we know that on a Mark XI, we know only that \texttt{findMin} will run between \( 0.6 \cdot F(n) \) and \( 1.6 \cdot F(n) \) microseconds. Then we know that on a Mark V, \texttt{findMin} will run between \( 0.6 \cdot 2.5 \cdot F(n) = 1.5 \cdot F(n) \) and \( 1.6 \cdot 3.8 \cdot F(n) = 6.08 \cdot F(n) \) microseconds — on reasonable size arrays.

The second assumption helps us figure out \( F(n) \) for a particular machine. This second assumption concerns *straightline code*. For our purposes straightline code is any section of Java code that does not contain a loop, recursion, or a call to method that somehow invokes a loop or a recursion. For example, in \texttt{findMin}

\[
\begin{align*}
\bullet & \ n = a.length; \\
& \minVal = a[0];
\end{align*}
\]

\[1\] There are some oversimplifications here that we’ll fix later.
• if (a[j]<minVal) {
    minVal = a[j];
}

are both examples straightline code, but

• for (j=1; j<n; j++)
    if (a[j]<minVal) { minVal = a[j]; }

is not straightline because of the for loop.

Assumption 2

Given:
• a particular method $M$,
• a particular section, $C$, of straightline code of method $M$,
• a particular set up, $S_1$, of computer, compiler, and runtime,
then there are positive (real) constants $c_{lower}$ and $c_{upper}$ such that

$$c_{lower} \leq \frac{\text{the } S_1 \text{ runtime of } C \text{ for size } n \text{ input}}{1 \text{ microsecond}} \leq c_{upper}$$

for any value of $n$.

Assumption 2 says that under a particular setup, the time to run a particular piece of straightline code does not depend on the input size. Assumption 2 also says that it is the loops and the recursions that cause the dependence on $n$ — because it sure ain’t the straightline code. So here is what we do to get $F(n)$, or at least an estimate of $F(n)$.

For the moment we will not worry about recursions, just loops.

1. We identify the pieces of straightline code that are buried deepest in the method’s loops. Example: In `findMin` the straightline code that is buried deepest is

   ```java
   if (a[j]<minVal) { minVal = a[j]; }
   ```

   since it occurs within the loop. Whereas

   ```java
   n = a.length; minVal = a[0];
   ```

   occurs outside the loop.

2. We count how many times these innermost pieces of the program are executed for a given value of $n$.

   Example: The loop of `findMin` is

   ```java
   for (j=1; j<n; j++)
       if (a[j]<minVal) { minVal = a[j]; }
   ```

   So, the first iteration has $j==1$, the last iteration has $j==n-1$, and $j$ increases by 1 with each iteration. So, there are $(n-1) - (1) +1 == n-1$ many iterations.

---

2 Again, we are oversimplifying things.
3. If the formula we get from step 2 is $G(n)$, then it follows from Assumption 2, that there are positive constants $c_{\text{lower}}$ and $c_{\text{upper}}$ such that the run time of our method is between $c_{\text{lower}} \cdot G(n)$ and $c_{\text{upper}} \cdot G(n)$.

Example: From step 2, our formula is $n - 1$. So, for the Woot™ Mark XI, there are positive constants $c_\ell$ and $c_u$ such that, for any array of length $n$,

$$c_\ell \cdot (n - 1) \ \mu\text{secs} \leq \text{runtime of } \text{findMin} \leq c_u \cdot (n - 1) \ \mu\text{secs}.$$

For a slightly different choice of positive constants $c'_\ell$ and $c'_u$, we have

$$c'_\ell \cdot n \ \mu\text{secs} \leq \text{runtime of } \text{findMin} \leq c'_u \cdot n \ \mu\text{secs}$$

for sufficiently large $n$. This last equality can be restated as:

$$c'_\ell \leq \frac{\text{the runtime of } \text{findMin}}{n \ \mu\text{secs}} \leq c'_u.$$

So, the runtime of $\text{findMin}$ is bounded below and above by linear functions (in $n$) on the Mark XI. By Assumption 1, the same is going to hold true (with different constants) on any other machine on which we run $\text{findMin}$. Because of this, say that $\text{findMin}$ has linear run time. If we want to know the constants for any particular machine, we can run some timing experiments on that machine.

6.3 A second example

Let us consider another example which is a bit more complex and a lot more interesting. Here is a Java version of the selection sort algorithm.

```java
public static void selectionSort(int a[]) {
    // PRECONDITION: a is not null.
    // POSTCONDITION: a is sorted in increasing order
    int i, j, minJ, minVal, n;
    n = a.length;

    for (i=0; i < n-1; i++) {
        // find a minJ ∈ {i, ..., n-1} such that
        // a[minJ] = min {a[j] : i ≤ j ≤ n-1}
        minJ = i; minVal = a[i];
        for (j=i+1; j<n; j++)
            if (a[j]<minVal) { minJ = j; minVal = a[j]; }

        // Swap the values of a[i] and a[minJ]
        a[minJ] = a[i]; a[i] = minVal;
    }
}
```
Now we have $a[0] \leq a[1] \leq \ldots \leq a[i]$

$\leq \min \{a[j] : i < j \leq n-1\}$

Selection sort works by first finding the smallest value in the array and making $a[0]$ equal to this value (by swapping the positions of some elements in the array). Then it finds the next smallest element in $a$ and makes $a[1]$ equal to this value (again by swaps). It keeps on doing this with $a[2], a[3], \ldots$ until it has a sorted array.

Let us analyze `selectionSort`'s runtime the same way we did with `findMin`.

1. **Identify the pieces of straightline code that are buried deepest.**

   In `selectionSort` this is:

   ```
   if (a[j]<minVal) { minJ = j; minVal = a[j]; }
   ```

   since it occurs within both loops. Whereas both

   - $\text{minJ = i; minVal = a[i];}$
   - $\text{a[minJ] = a[i]; a[i] = minVal;}$

   occur within only the outermost loop.

2. **Count how many times these innermost pieces of the program are executed for a given value of $n$.**

   We handle the two loops in `selectionSort` one at a time, starting with the innermost.

   **The innermost loop**: This is

   ```
   for (j=i+1; j<n; j++)
     if (a[j]<minVal) {minJ = j; minVal = a[j];}
   ```

   So, the first iteration has $j=i+1$, the last iteration has $j=n-1$, and $j$ increases by 1 with each iteration. So, there are $(n-1) - (i+1) + 1 = n-i-1$ many iterations. So for particular values of $i$ and $n$ the innermost code is executed $n-i-1$ times every time the innermost for loop is executed.

   **The outermost loop**: This is

   ```
   for (i=0; i<n-1; i++) { 
     minJ = i; minVal = a[i];
     — the innermost loop —
     a[minJ] = a[i]; a[i] = minVal;
   }
   ```

   The first iteration thus has $i = 0$, the last iteration has $i = n - 2$, and $i$ increases by 1 with each iteration. So, the number of times the innermost code is executed is:
<table>
<thead>
<tr>
<th>iteration #</th>
<th>value of i</th>
<th># of executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>n-1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>n-2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>n-3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n-3</td>
<td>n-4</td>
<td>3</td>
</tr>
<tr>
<td>n-2</td>
<td>n-3</td>
<td>2</td>
</tr>
<tr>
<td>n-1</td>
<td>n-2</td>
<td>1</td>
</tr>
</tbody>
</table>

Therefore, the total number of executions is

\[ 1 + 2 + 3 + \cdots + (n - 1). \]

By a standard bit of math, this last sum equals

\[ \frac{(n - 1) \cdot n}{2} = \frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n. \]

3. **Take the count from step 2 and conclude the “order” of the runtime on any particular** **machine.**

So for our faithful Mark XI, there are positive constants \( c_\ell \) and \( c_u \) such that, for any array of length \( n \),

\[ c_\ell \cdot (\frac{1}{2}n^2 - \frac{1}{2}n) \mu \text{secs} \leq \text{runtime of selectionSort} \leq c_u \cdot (\frac{1}{2}n^2 - \frac{1}{2}n) \mu \text{secs}. \]

For a slightly different choice of positive constants \( c'_\ell \) and \( c'_u \), we have

\[ c'_\ell \cdot n^2 \mu \text{secs} \leq \text{runtime of selectionSort} \leq c'_u \cdot n^2 \mu \text{secs} \]

for sufficiently large \( n \).

So, the runtime of selectionSort is bounded below and above by quadratic functions (in \( n \)) on the Mark XI. By Assumption 1, the same is going to hold true (with different constants) on any other machine on which we run selectionSort. Because of this, say that selectionSort has \textit{quadratic run time}.

The examples show that we need a way of talking about run times (or functions in general) that have a linear or quadratic growth rate — without having to bring in the irritating constants all the time.

**Convention:** Let \( f \) and \( g \) range over functions from \( \mathbb{N} = \{0, 1, 2, \ldots\} \) to \( \mathbb{N}^+ = \{1, 2, 3, \ldots\} \).

The collection of functions that have linear growth rate is the set:

\[ \left\{ f \mid \text{for some } c_\ell, c_u > 0, \quad c_\ell \leq f(n)/n \leq c_u \right\} \]
Let us give this set the somewhat funny name $\Theta(n)$. So we can now say that \texttt{findMin} has a run time in $\Theta(n)$.

The collection of functions that have quadratic growth rate is the set:

$$
\{ f | \text{ for some } c_\ell, c_u > 0, \quad c_\ell \leq f(n)/n^2 \leq c_u \}
$$

Let us give this set the somewhat funny name $\Theta(n^2)$. So we can now say that \texttt{selectionSort} has a run time in $\Theta(n^2)$.

In general, the collection of functions that have growth rate like some given function $g$ is the set:

$$
\{ f | \text{ for some } c_\ell, c_u > 0, \quad c_\ell \leq f(n)/g(n) \leq c_u \}
$$

We give this set the name $\Theta(g(n))$.

With this $\Theta$ notation we can talk in general about the growth rate of functions.

**Example.** Suppose $f$ is given by:

$$
f(n) = 100n^2 + 8000n + 97.
$$

We claim that $f$ is in $\Theta(n^2)$. We can do this by doing a bit of high-school algebra and show

$$
100 \leq f(n)/n^2 \leq 200
$$

for all $n > 81$. Hence, $f$ belongs to the $\Theta(n^2)$ club.

**Example.** Suppose $f$ is given by:

$$
f(n) = 8000n + 97.
$$

We claim that $f$ is not in $\Theta(n^2)$. To see this, suppose that there are $c_\ell, c_u > 0$ such that

$$
c_\ell \leq \frac{8000n + 97}{n^2} \leq c_u
$$

for all sufficiently large $n$. But this can’t be since by choosing $n$ large enough we can make

$$
\frac{8000}{n} + \frac{97}{n^2} \leq c_\ell.
$$

So no such $c_\ell$ can exist.

There is a fast way to do examples such as those above.
The Limit Rule (Version 1) Suppose that

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \]

where \( c \) is such that \( 0 \leq c \leq +\infty \).

(a) If \( 0 < c < +\infty \), then \( f \) is in \( \Theta(g(n)) \).
(b) If \( c = 0 \) or \( c = \infty \), then \( f \) is not in \( \Theta(g(n)) \).

For example,

\[ \lim_{n \to \infty} \frac{100n^2 + 8000n + 97}{n^2} = 100. \]

Therefore, \( 100n^2 + 8000n + 97 \) is in \( \Theta(n^2) \). Also

\[ \lim_{n \to \infty} \frac{8000n + 97}{n^2} = 0. \]

Therefore, \( 8000n + 97 \) is not in \( \Theta(n^2) \).

6.4 A third example

Now let us deal with some of the oversimplifications we confessed to above. Another sorting algorithm will help show what the problem is. Notation: \( a[i..j] \) refers to “subarray” of \( a \): \( a[i], a[i+1], \ldots, a[j] \).

```java
public static void insertionSort(int a[]) {
    // PRECONDITION: a is not null.
    // POSTCONDITION: a is sorted in increasing order
    int i, j, key, n;
    n = a.length;

    for (i=1; i < n; i++) {
        // We assume a[0..(i-1)] is already sorted.
        // We save the initial value of a[i] in key.
        // Each value in a[0...(i-1)] that is larger
        // than key is shifted up one position.
        // This opens a hole between the values \leq key
        // and the values \geq key.
        // We place the value in key in this hole.
        key = a[i]; j = i-1;
        while (j>=0 && a[j]>key) {
            a[j+1] = a[j]; j = j -1;
        }
        a[j+1] = key;
        // Now a[0] \leq a[1] \leq \cdots \leq a[i].
    }
}
```
Insertion sort has the property that it is much faster on some inputs than others. Here are
the two key examples.

1. Suppose the array to be sorted is already in increasing order. Then in every iteration
of the for-loop, the while-loop test $a[j] > key$ will fail the first time we try it. So, we go
through no iterations of the while-loop at all and a little work shows that on already
sorted inputs, \texttt{insertionSort} runs in $\Theta(n)$ time.

2. Suppose the array to be sorted is already in decreasing order (that is, from biggest to
smallest). Then in every iteration of the for-loop, the while-loop has to go through
$i$-many iterations since initially all of the values in $a[0], \ldots, a[i-1]$ are larger than
\texttt{key}. So an analysis similar to that for \texttt{selectionSort} shows that for inputs that are
sorted in decreasing order, \texttt{insertionSort} runs in $\Theta(n^2)$ time.

Hence, for a particular algorithm, inputs of the same size can result in quite different run
times.

**Terminology:** The best case run time of an algorithm (or program or method) on an input
of size $n$ is the best (i.e., smallest) run time of the algorithm on size-$n$ inputs. The worst
case run time of an algorithm (or program or method) on an input of size $n$ is the worst
(i.e., biggest) run time of the algorithm on size-$n$ inputs. For example,

- \texttt{findMin} has linear best and worse case run times.
- \texttt{selectionSort} has quadratic best and worse case run times.
- \texttt{insertionSort} has linear best case and quadratic worse case run times.

To deal with the fact that the order of the best and worse case run times may not match,
we introduce ways of taking about upper and lower bounded on growth rates.

### 6.5 Upper and lower bounds on rates of growth

Recall that

\[
\Theta(g(n)) = \{ f | \text{ for all sufficiently large } n \} 
\]
\[
\text{for some } c_\ell, c_u > 0, 
\]
\[
c_\ell \leq f(n)/g(n) \leq c_u
\]

To talk about upper and lower bounds on rates of growth, we break $\Theta$ in two parts. For a
given $g$, define

\[
O(g(n)) = \{ f | \text{ for all sufficiently large } n \} 
\]
\[
\text{for some } c_u > 0, 
\]
\[
f(n)/g(n) \leq c_u
\]

\[
\Omega(g(n)) = \{ f | \text{ for all sufficiently large } n \} 
\]
\[
\text{for some } c_\ell > 0, 
\]
\[
c_\ell \leq f(n)/g(n)
\]
Intuitively:

- $\Theta(g(n))$ is the collection of functions that have growth rates proportional to $g(n)$.
- $O(g(n))$ is the collection of functions $f(n)$ such that $f(n)/g(n)$ is no worse than some constant for large $n$.
- $\Omega(g(n))$ is the collection of functions $f(n)$ such that $f(n)/g(n)$ is no smaller than some positive constant for large $n$.

**Theorem.** $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$. That is,

$$f \in \Theta(g(n)) \text{ if and only if } f \in O(g(n)) \text{ and } f \in \Omega(g(n)).$$

**Example.** Suppose $f$ is given by:

$$f(n) = 8000n + 97.$$ 

We claim that $f$ is in $O(n^2)$. To see this, use some high school math to show

$$8000n + 97 \leq 8000 \cdot n^2$$

for all $n > 3$.

**Example.** Suppose $f$ is given by:

$$f(n) = 100n^2 + 8000n + 97.$$ 

We claim that $f$ is in $\Omega(n)$. To see this, use some high school math to show

$$n \leq 100n^2 + 8000n + 97$$

for all $n > 0$.

We can extend the limit rule to help with these $O$ and $\Omega$ problems.

**The Limit Rule (Version 2)** Suppose that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

where $0 \leq c \leq +\infty$.

(a) If $0 < c < +\infty$, then $f$ is in $\Theta(g(n))$, $O(g(n))$, and $\Theta(g(n))$.

(b) If $c = 0$, then $f$ is in $O(g(n))$, but **not** in either $\Theta(g(n))$ or $\Omega(g(n))$.

(c) If $c = \infty$, then $f$ is in $\Omega(g(n))$, but **not** in $\Theta(g(n))$ or $O(g(n))$.  

25
6.6 Take-home exam problems

Problem 6.1: Take Home Exam Problem 1: Due by email on Friday Dec 7th

For each ordered pair of functions in the list below, determine all pairs \( f(n), g(n) \) such that \( f(n) = O(g(n)) \) and whether for each such pair, \( f(n) = \Theta(g(n)) \). By definition \( f(n) = o(g(n)) \) if, and only if, \( f(n) = O(g(n)) \) and \( f(n) \neq \Theta(g(n)) \). The “little oh” relationship, \( o \), determines an ordering among these functions. Draw a graph showing this ordering. Then draw a graph showing the ordering among the \( \Theta \) equivalence classes of these functions.

\[
\begin{align*}
\sqrt{2}^{\log_2 n} \\
n^2 \\
n! \\
\log_2 n! \\
3^n \\
2 \\
n^3 \\
\log_2(\log_2 n) \\
2^{2^n} \\
n^{1/\log_2 n} \\
n \cdot 2^n \\
n^{\log_2(\log_2 n)} \\
1
\end{align*}
\]
\[2^{\log_2 n}\]

\[(\log_2 n)^{\log_2 n}\]

\[2^n\]

\[4^{\log_2 n}\]

\[(n + 1)!\]

\[\sqrt{\log_2 n}\]

\[\triangle\]

**Problem 6.2: Take Home Exam Problem 2: Due by email on Friday Dec 7th**

On the next page there are diagrams depicting pairs of tree rotations to be performed as elements are inserted into trees. The basic tree rotations are shown in the diagrams immediately below.

The preceding diagram as well as the diagram on the following page are from the Wikipedia article on AVL-trees.
Show the results of beginning with the empty tree and inserting with rebalancing after each insertion the numbers 3, 1, 4, 5, 8, 0, 2, 6, 9, 7 in the order given. Then show the results, with rebalancing after each deletion, of removing 4 and then 0.
Consider the following quantum program code.

\[
\text{qubit } p, q, r; \quad // \text{All qubits are assumed to be randomly}
\]
\[
\text{initialized in some quantum state.}
\]

With this declaration the states of \( p, q \) and \( r \) are

\[
a|0\rangle + b|1\rangle, \quad c|0\rangle + d|1\rangle \text{ and } e|0\rangle + f|1\rangle
\]

respectively, where \( a, b, c, d, e \) and \( f \) are complex numbers that satisfy

\[
|a|^2 + |b|^2 = 1
\]
\[
|c|^2 + |d|^2 = 1
\]
\[
|e|^2 + |f|^2 = 1
\]

In the quantum state of qubit, \( \alpha|0\rangle + \beta|1\rangle \), the coefficients \( \alpha \) and \( \beta \) are called \textit{amplitudes}.

In this problem, all of the amplitudes we encounter will be real numbers, so you need not worry about the details of complex numbers. The state of the combined system of all three variables is

\[
(a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) \otimes (e|0\rangle + f|1\rangle)
\]

The operation \( \otimes \), called \textit{tensor}, works like ordinary multiplication:

\[
(a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) \otimes (e|0\rangle + f|1\rangle)
\]

\[
= \left( (a|0\rangle + b|1\rangle) \otimes c|0\rangle + (a|0\rangle + b|1\rangle) \otimes d|1\rangle \right) \otimes (e|0\rangle + f|1\rangle)
\]

\[
= (a|0\rangle c|0\rangle + b|1\rangle c|0\rangle + a|0\rangle d|1\rangle + b|1\rangle d|1\rangle) \otimes (e|0\rangle + f|1\rangle)
\]

\[
= (ac|0\rangle|0\rangle + bc|1\rangle|0\rangle + ad|0\rangle|1\rangle + bd|1\rangle|1\rangle) \otimes (e|0\rangle + f|1\rangle)
\]

\[
= ace|0\rangle|0\rangle|0\rangle + bce|1\rangle|0\rangle|0\rangle + ade|0\rangle|1\rangle|0\rangle + bde|1\rangle|1\rangle|0\rangle +
\]
\[
acf|0\rangle|0\rangle|1\rangle + bcf|1\rangle|0\rangle|1\rangle + adf|0\rangle|1\rangle|1\rangle + bdf|1\rangle|1\rangle|1\rangle
\]

Now, this is very, very important: Suppose that \( U \) is an allowed operation on a qubit’s quantum state other than a measurement. Such an operation is called a 1-qubit unitary operation. We need not be concerned in this problem precisely what these operations are because we will only use a few of them and we will be explicit about what they are. Here is one example: Let

\[
H(|0\rangle) = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle
\]

and let

\[
H(|1\rangle) = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle
\]
$H$ is called the Hadamard operation and is one of these allowed unitary operations. Any such allowed operation $U$ extends to arbitrary qubit states like this:

$$U(\alpha|0\rangle + \beta|1\rangle) = \alpha U(|0\rangle) + \beta U(|1\rangle)$$

$U$ can be extended this way because $U$ is a linear operation. Being linear is implied by being unitary. So according to the rules so far, for example,

$$H(a|0\rangle + b|1\rangle) = a(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) + b(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle)$$

$$= a + b|0\rangle + \frac{a - b}{\sqrt{2}}|1\rangle$$

What happens if the operation is performed on a qubit that is part of a system of qubits such as in the code we are examining? What for example, happens if we apply the Hadamard operation to qubit $p$ (the first qubit in each term corresponds to $p$) as part of the state

$$ace|0\rangle|0\rangle|0\rangle + bce|1\rangle|0\rangle|0\rangle + ade|0\rangle|1\rangle|0\rangle + bde|1\rangle|1\rangle|0\rangle +$$

$$acf|0\rangle|0\rangle|1\rangle + bcf|1\rangle|0\rangle|1\rangle + adf|0\rangle|1\rangle|1\rangle + bdf|1\rangle|1\rangle|1\rangle$$

The answer is that we apply the operation to the $p$ qubit value in each term separately and
then substitute. After that, rearranging the sum to simplify helps:

\[
H(ace|0\rangle|0\rangle|0\rangle + bce|1\rangle|0\rangle|0\rangle + ade|0\rangle|1\rangle|0\rangle + bde|1\rangle|1\rangle|0\rangle + \\
acf|0\rangle|0\rangle|1\rangle + bcf|1\rangle|0\rangle|1\rangle + adf|0\rangle|1\rangle|1\rangle + bdf|1\rangle|1\rangle|1\rangle)
\]

\[
= ace\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right)|0\rangle|0\rangle + bce\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)|0\rangle|1\rangle + \\
adf\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)|1\rangle|0\rangle + bde\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)|1\rangle|1\rangle + \\
acf\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right)|0\rangle|1\rangle + bcf\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)|0\rangle|1\rangle + \\
acf\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right)|1\rangle|1\rangle + bdf\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)|1\rangle|1\rangle
\]

\[
= \left(\frac{a + b}{\sqrt{2}}\right)ce|0\rangle|0\rangle|0\rangle + \left(\frac{a - b}{\sqrt{2}}\right)ce|1\rangle|0\rangle|0\rangle + \left(\frac{a + b}{\sqrt{2}}\right)de|0\rangle|1\rangle|0\rangle + \\
\left(\frac{a - b}{\sqrt{2}}\right)de|1\rangle|1\rangle|0\rangle + \left(\frac{a + b}{\sqrt{2}}\right)cf|0\rangle|0\rangle|1\rangle + \left(\frac{a - b}{\sqrt{2}}\right)cf|1\rangle|0\rangle|1\rangle + \\
\left(\frac{a + b}{\sqrt{2}}\right)df|0\rangle|1\rangle|1\rangle + \left(\frac{a - b}{\sqrt{2}}\right)df|1\rangle|1\rangle|1\rangle
\]

So, to summarize: one applies the operation to the appropriate qubit in each individual term and then simplifies. (You may notice a pattern in the above display that will save you some time when you do these sorts of calculations.)

For the next part of the code we need to understand how measurement works. When we execute a measure operation on a qubit two things happen: (1) the qubit changes state, and (2) a bit value is returned: If we measure a qubit in quantum state \(\alpha|0\rangle + \beta|1\rangle\), then 0 will be returned with probability \(|\alpha|^2\) and 1 will be returned with probability \(|\beta|^2\). If 0 is returned, the state of the qubit jumps to 

\[1|0\rangle + 0|1\rangle\]

which we can write simply as 

\[|0\rangle\]

Similarly, if 1 is returned, the state of the qubit jumps to \(|1\rangle\). (By the way, that’s the kind of thing a real quantum jump is.) What, for example, happens when we measure, say, qubit
r (the right-hand qubit) in the state

\[ ace|0⟩|0⟩|0⟩ + bce|1⟩|0⟩|0⟩ + ade|0⟩|1⟩|0⟩ + bde|1⟩|1⟩|0⟩ + \]

\[ acf|0⟩|0⟩|1⟩ + bcf|1⟩|0⟩|1⟩ + adf|0⟩|1⟩|1⟩ + bdf|1⟩|1⟩|1⟩ \]

and 1 is returned? We drop all terms with \(|0⟩\) (corresponding to the value not returned) corresponding to qubit \(r\) to obtain:

\[ acf|0⟩|0⟩|1⟩ + bcf|1⟩|0⟩|1⟩ + adf|0⟩|1⟩|1⟩ + bdf|1⟩|1⟩|1⟩ \]

But there is a problem: the squares of the absolute values of the amplitudes probably don’t add up to 1, and they must in any quantum state. So, one multiplies each term by a scalar to restore the property that the amplitudes add up to 1. Fortunately, calculating the scalar is easy. It’s

\[ \frac{1}{\sqrt{p}} \]

where \(p\) is the probability that is associated with the result returned by the measurement.

\[ p = |acf|^2 + |bcf|^2 + |adf|^2 + |bdf|^2 = (|ac|^2 + |bc|^2 + |ad|^2 + |bd|^2)|f|^2 = |f|^2 \]

Another example: Suppose a system of two qubits is in state

\[ \frac{1}{\sqrt{2}} |0⟩|0⟩ + \frac{1}{\sqrt{2}} |1⟩|1⟩ \]

Suppose we measure the right-hand qubit and 0 is returned. The system jumps to

\[ \frac{1}{\sqrt{\text{Prob}(0 \text{ is returned})}} \frac{1}{\sqrt{2}} |0⟩|0⟩ = |0⟩|0⟩ \]

since the probability that a 0 is returned is \( \left| \frac{1}{\sqrt{2}} \right|^2 = \frac{1}{2} \).

When a command `measure k in x` is executed, where \(k\) is a qubit variable and \(x\) is a bit variable, a bit value is returned and assigned to \(x\). (Physical reality really behaves in this probabilistic way.) However you calculate the necessary scalar to restore the amplitudes summing to 1, the scalar is a nonnegative real number. This is called renormalization, and you must renormalize after a measurement.

Continuing the program code:

```
bit u,v,w; //u,v,w are initialized to 0.

measure q in w; // After execution the state of q and v
// is either |0> and 0, or |1> and 1.
```

Next we are going to perform a branching if-then-else command. After the branching command executes, the quantum state of qubit \(q\) will be \(|0⟩\).
if (w = 0) // Notice that the state transition performed by
// this branching command is not unitary.
then
I(q) // I(j) is the identity operation on qubit j.
// It’s a no-op.
else
NOT(q); // The state of q after this branching command is executed
// is |0>. NOT(x|0> + y|1>) = y|0> + x|1>
measure r in w;
if (w = 0)
then
I(r)
else
NOT(r); // At this point the state of the qubit pair (q,r) is
// |0>|0>. The combined state of all three qubits is
// a|0>|0>|0> + b|1>|0>|0>
Bell(q,r); // Bell(j,k) can be any 2-qubit unitary that operates
// on qubits j and k by mapping |00> to the so-called
// Bell state 1/sqrt{2}|0>|0> + 1/sqrt{2}|1>|1>.
// To calculate the combined state of all three qubits,
// multiply out
// (a|0> + b|1>)
// tensor (1/sqrt{2}|0>|0> + 1/sqrt{2}|1>|1>)
// and simplify.
CNOT(p,q); // CNOT(|0>|0>) = |0>|0>. CNOT(|0>|1>) = |0>|1>.
// CNOT(|1>|0>) = |1>|1>. CNOT(|1>|1>) = |1>|0>.

**DO:** Calculate the combined state of the system of all three qubits.
Hadamard(p);

measure p in u;
measure q in v; // Four possibilities for u, v

if (u = 0) and (v = 0) then
  I(r) //Calculate the quantum state of r
else if (u = 0) and (v = 1) then
  NOT(r) //Calculate the quantum state of r
else if (u = 1) and (v = 0) then
  PhaseFlipPi(r) //Calculate the quantum state of r
    //PhaseFlipPi(x|0> + y|1>) = x|0> - y|1>
else {
  PhaseFlipPi(r);
  NOT(r) //Calculate the quantum state of r

DO: the calculations in the last four comments in the code above.

Problem 6.4: Take Home Exam Problem 4: Due by email on Wednesday Dec 12th
The following data structure is often referred to as a hash table.

The mapping from keys to “buckets”, i.e. array indices, is carried out by a hash function. There are a variety of approaches to a hash function but many take advantage of easy-to-compute number-theoretic functions that mix-up their outputs in a pseudo-random manner. Let σ be a character string. Let nat(σ) be the result of interpreting the character string σ
as a nonnegative integer. For example, “cat” as the concatenation of a left-most 1 with the ascii codes of its three characters is, as an integer in expressed in decimal, 1099097116. Let

\[ h(n) = \text{floor}(2^8 \cdot ((0.123456 \cdot n) \mod 1)) \]

Thus, for example,

\[ h(\text{nat(“cat”)}) = h(1099097116) = \text{floor}(2^8 \cdot ((0.123456 \cdot 1099097116) \mod 1)) = 141 \]

Thus a pointer to the data whose key is “cat” would be inserted into the array at cell 141. If later on, data whose key was “qGb” is inserted, we have

\[ h(\text{nat(“qGb”)}) = h(1113071098) = \text{floor}(2^8 \cdot ((0.123456 \cdot 1113071098) \mod 1)) = 141 \]

The hash value produces a collision of “qGb” with “cat”. Thus the data for which “qGb” is the key would be \textit{cons} onto the list pointed to at array cell 141.

\textbf{The Problem:} Suppose we have a hash table with 4 array cells, indexed from 0 to 3. Let the hash function of the data structure be given by

\[ h(n) = \text{floor}(2^2 \cdot ((0.17 \cdot n) \mod 1)) \]

Insert, in the order given, the following character strings: “dec”, “pr”, “o”, “b”, “le”, “m”, “set”. Show the state of the hash table after each insertion.

\[ \triangle \]

\textbf{Problem 6.5: Take Home Exam Problem 5: Due by email on Tuesday Dec 11th}

This problem is about the \textit{array implementation of binary max-heaps}. An array implementation of a binary tree (any kind) works like as follows. In class I indexed the array that will contain the binary tree by starting at 1. In this problem I will assume that array indices start at 0. The root of the binary tree is stored in cell 0. In general, if a given vertex \( v \) of the tree is stored in cell \( k \), then the left child of \( v \) is stored in cell \( 2k + 1 \) and the right child is stored in cell \( 2k + 2 \). For example, suppose we have a binary tree whose indices contain the five keys 2, 3, 7, 17, 19 like this:

Then an array implementation of this tree with 7 ((a sufficiently large power of 2) - 1) or more cells would be filled in like this:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
19 & 17 & 7 & 2 & 3 & 19
\end{array}
\]
To use a binary tree as a binary heap we must, by definition, fill in each level of the tree from left to right. So the tree above should have the shape below in order to be eligible to be a heap:

![Binary Tree Diagram]

The corresponding array is

```
  0 1 2 3 4 5 6
19 17 7 2 3 25
```

The requirement to fill in each level of the tree from left to right assures there will be no gaps in the array that implements the tree.

Max heaps have an additional requirement: The key in each node must be greater than the key in each child of the node. So, suppose we want to insert a node with key 100. Since the tree must filled in on each level from left to right, we must insert 100 as shown in the following array:

```
  0 1 2 3 4 5 6
19 17 100 2 3 25
```

But 100 is now a child of 7, which violates the Max heap property. To restore the max heap property we “bubble-up” the node toward the root until the max heap property is restored by repeated swaps of the node with key 100 with its parent.

```
  0 1 2 3 4 5 6
19 17 100 2 3 7
```

Suppose we extend this array out to 15 cells:

```
  0 1 2 3 4 5 6
19 17 100 2 3 7
```

Show the array after inserting 25, 36, and 1 in that order. During each insertion show the array after each swap individual swap during the bubble-up to restore the max heap property.

Problem 6.6: Take Home Exam Problem 6: Due by email on Wednesday Dec 12th Suppose that A is a 100000 by 100000 2-dimensional array of booleans in which every cell is filled in with a Boolean value. Describe a method to produce a 1-dimensional array
\( B \) with 100000 cells that is completely filled in with Boolean values in such a way that \( B \) is different from every row and every column of \( A \).