Problem Set 1: Making Mosaics

Due: Monday, 22 January
Beginning of class

Turn-in Checklist: On 22 January, bring to class a stapled turn in containing your answers to questions 1-5. Your answers should be printed in order and formatted clearly. Include all the code you wrote, but do not include large amounts of code we provided in your turn in.

Collaboration Policy - Read Carefully

For this assignment, you may work alone or with one partner of your choice. If you work with a partner, you and your partner should turn in one problem set with both of your names on it. Before you start working with your partner, you should read through the assignment yourself and think about the questions.

You may consult any outside resources including books, papers, web sites and people, you wish except for materials from previous CS150 courses. If you use resources other than the class materials, indicate what you used along with your answer.

You are **strongly encouraged** to take advantage of the staffed lab hours in Small Hall:

Thursday 6-9pm
Friday 1-2:30pm (right after class)
Sunday 4-5:30pm, 8-9:30pm

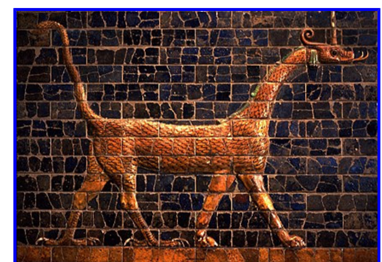
Purpose

- Introduce *divide and conquer* problem solving.
- Provide exposure to recursive definitions and functions as parameters.
- Provide experience reading a Scheme program.
- Learn to use and create functions.
- Introduce the DrScheme environment and Scheme programming language.
- Make a pretty picture.

Warning: This problem set is different from typical assignments. It touches on lots of new concepts, but you are not expected to understand everything in this assignment yet. It is recommended that you skim through the whole problem set before trying to answer the questions so you have a clear idea on what you need to do. You should attempt to understand as much of the rest as possible, but don't worry if it doesn't all make sense yet. We won't cover everything you need to understand all the provided code until the end of the course.

Background

A mosaic is a picture made up of lots of smaller pieces. The ancient Babylonians constructed elaborate mosaics using colored tiles. A *photo*mosaic is a mosaic, except instead of using single color tiles for the tiles, it uses photographs. For some example photomosaics, see *http://www.photomosaic.com/*.



Making a photomosaic is a big task. The way computer scientists (and most other people) solve big problems, is to break them into a few smaller sub-problems whose solutions can be combined to solve the original problem. This approach is sometimes called *divide and conquer*.

Note that for a big problem like making a photomosaic, doing this once is probably not enough. The sub-problems are still too big to solve easily. So, we need to use divide-and-conquer again to break each sub-problem into several sub-sub-problems. We do this recursively, until we get to sub-sub-sub-...-sub problems that are small enough that we can solve them easily.

We can describe this process with *pseudocode* (that is, not real Scheme, but made to look like it):

```
(define (solve-big-problem problem)
  (if (easy-to-solve? problem)           ; if the problem is easy to solve,
      (solve problem)                    ;    just solve it!
      (combine-solutions                 ; otherwise, combine the solutions
        (map solve-big-problem           ;    of solving
             (divide-problem problem))))) ;       the sub-problems you
                                         ;       get by dividing the
                                         ;       original problem
```

Here, `map` is a function that applies another function to every item in a list. The `(divide-problem problem)` evaluates to a list of sub-problems. The value of `(map solve-big-problem (divide-problem problem))` is the result of applying `solve-big-problem` to every sub-problem in that list.

Implementing `solve`, `combine-solutions` and `divide-problem` for a real problem is the difficult part. There is a real art to figuring out how to divide a big problem into a suitable set of smaller problems. Being able to do this well is the main thing that separates good problem solvers from mediocre ones.

Reading: Before going further, you should have finished reading Chapters 2 and 3 from the course book.

Scheme Expressions

Question 1: For this question you should not use the Scheme interpreter. For each fragment below, either:

1. Explain why the fragment is not a valid Scheme expression; or,
2. Predict what value the expression will evaluate to.

a. `150`
b. `(+ 70 80)`
c. `+`
d. `(100 + 100)`
e. `(> 150 100)`
f. `(and (> 3 2) (> 4 5))`
g. `(if (> 12 10) "good move" "try again")`
h. `(if (not "cookies") "eat" "starve")`

Lab: These questions involve using a computer. Make sure to take turns driving (who is typing on the keyboard). You and your partner should switch positions every few minutes. Although only one partner can type at a time, both partners should be *thinking* all the time!

Question 2: Try evaluating all the expressions from Question 1 using DrScheme. For instructions for running DrScheme, see the Lab Guide. Remember especially to set the language to `Pretty Big` as described in the Lab Guide. Turn in a print out of your interactions buffer with your explanations either as handwritten notes or as Scheme comments (everything after the semi-colon) in your printout. You can generate this by selecting `File | Print Interactions` from the DrScheme menu. If any expression evaluates to something different from what you expected in Question 2, figure out why. Don't change your answers to Question 2, but write down explanations of why you think DrScheme produces the value it does instead of what you expected. (You don't lose any points on Question 2 if your answers are wrong.)

Photomosaics

A (not too brilliant) kindergartner could follow these directions to make a photomosaic:

1. Collect pictures to use as the tiles
    a. Ask your parents for some old magazines and scissors. (*Note: it is important that you let your parents know you will cut up the magazines!*)
    b. Repeat many times until you have enough pictures:
        i. Pick one of the magazines.
        ii. Look through it until you find a good picture. A good picture is pretty small and colorful.
        iii. Cut out the picture using the scissors.
2. Break for milk and cookies.
3. Find a really big picture you want to use as the master for the photomosaic.
4. Put a thin piece of tracing paper on top of the master picture.
5. Draw a grid on the tracing paper:
    a. Get a ruler and crayon.
    b. Repeat until the whole paper is covered with lines, starting at the left edge:
        i. Line up the ruler parallel (*Note: you might have to explain what this means to the kindergartener, but you can always give her a copy of Euclid's Elements*) to the long edge of the paper.
        ii. Draw a line using the ruler.
        iii. Move the ruler a little bit to the right.
    c. Repeat until the whole paper is covered with lines, starting at the top:
        i. Line up the ruler parallel to the top edge of the paper.
        ii. Draw a line using the ruler.
        iii. Move the ruler a little bit down the page.
6. Naptime.
7. Put on the tiles:
    a. For each rectangle on the grid that you drew on the tracing paper:
        i. Look through the tile pictures to find one that best matches the color on the master picture under that rectangle.
        ii. Glue that tile picture onto the rectangle.
8. Clean up the glue you spilled on the floor before Mommy gets home.

We will create a photomosaic using Scheme in almost the same way — except the computer is much dumber than our kindergartener, so we need to break the steps into even smaller problems before it can perform them. (Our computer won't need to break for milk and cookies, but it will need to collect garbage.)

In step 7.a.i. the kindergartner has to look through the tile pictures to find one that *best matches the color on the master picture under that rectangle*. Your task for this problem set will be to define a function that does that.

First, we explain how the rest of the photomosaic program works. We show lots of code here, but **don't worry** if you don't understand all of it. You should attempt to understand how it works, but it is not necessary to completely understand all the code shown here to complete this assignment.

Creating a Photomosaic

We can divide the photomosaic problem into three big steps:

1. Get images for the tiles and master.
2. Select the tile images that best match the colors on the master.
3. Display the tile images.

First, we provide some background on images. Then we solve step 2 first, and then steps 1 and 3. Often, it is easier to solve a problem by considering the sub-problems out of order.

Download: Download *ps1.zip* to your machine and unzip it into your home directory `J:\cs150\ps1` (it will take some time to download because it contains many tile images, so start the download and continue reading while it is finishing). See the Lab Guide document for instructions on unziping files and creating directories.

This file contains:

- *ps1.ss* - A template for your answers. You should do the problem set by editing this file.
- *mosaic.ss* - Scheme code for producing photomosaics
- *images* - A folder containing the tile images you will use to make your photomosaic

You should look at the *mosaic.ss* (but don't worry if you don't understand much of the code in that file), but you will only need to change and turn in *ps1.ss*. If you double-click on `ps1.ss`, it should open in DrScheme. Click the `Execute` button (or use `ctrl+T`) to interpret the definitions. DrScheme will split into two windows, with an Interaction Window on the bottom. If the language is set correctly, you should see:

> Welcome to DrScheme, version 205.
> Language: Pretty Big (includes MrEd and Advanced).
> >

in your Interactions Window.

Color

All colors can be created by mixing different amounts of red, green and blue light — if you look closely at a television, you will see the whole image is made up of red, green and blue dots. We can represent a color using three numbers representing how much red, green and blue is in the pixel. For most computer images, there are a limited number of colors. In the images we will use, there are 256 different intensity values for each color. Hence, we can represent a color using three values between 0 and 255 to represent the amount of red, green, and blue in a pixel.

For example, `(0 0 0)` is black, `(255 255 255)` is white, `(255 0 0)` is red, `(0 0 255)` is blue, and `(255 255 0)` is yellow (mixing red and green light makes yellow).

The file `mosaic.ss` (included in the *ps1.zip* file you downloaded) defines some procedures for manipulating colors:

- `(make-color red green blue)` - evaluates to a color with red, green and blue components given by the parameters. The color values are between 0 and 255. For example, `(make-color 255 0 0)` is red.
- `(get-red color)` — evaluates to the red component of the color parameter. For example, `(get-red (make-color 255 0 0)) ==> 255`.
- `(get-green color)` — evaluates to the green component of the color parameter.
- `(get-blue color)` — evaluates to the blue component of the color parameter.
- `(show-color color)` — pops up a window that displays a rectangle of the color passed as its parameter.

It also defines some standard colors including:

```
(define white (make-color 255 255 255))
(define black (make-color 0 0 0))
(define red (make-color 255 0 0))
(define green (make-color 0 255 0))
(define blue (make-color 0 0 255))
(define yellow (make-color 255 255 0))
```

In your interaction buffer, try evaluating `(show-color red)`. You should see a small window appear containing a red rectangle.

> **Question 3:** Define a new color `orange` that looks like orange. If you can't find the color you want, you can find a chart of the RGB (red, green and blue) values of some popular colors at *http://www.lynda.com/hexh.html*. Use the `show-color` procedure to check that the color you defined looks like orange.

We can say color 1 is *brighter* than color 2 if the sum of the red, green and blue values of color 1 is greater than the sum of the red, green and blue values of color 2. This may not correspond exactly to our intuitive perception of colors, but it should be pretty close.

> **Question 4:** Define a procedure `brighter?` that takes two colors as parameters and evaluates to true (`#t`) if the first color is brighter than the second color; otherwise, it evaluates to false. Your procedure should look like:
>
> ```
> (define (brighter? color1 color2)
>    ;;; Fill in the code here that determines if color1 is brighter than color2
>    ;;; A good definition will need only a few lines of code.
>    )
> ```

Test your procedure by evaluating it with several different parameters. For example, you should get these results:

```
> (brighter? white red)
#t
> (brighter? red white)
#f
> (brighter? red red) ;; red is as bright as red, but not brighter
#f
> (brighter? red blue)
#f
> (brighter? yellow red)
#t
```

Images

A computer represents an image by a *bitmap*. A bitmap is a two-dimensional array of picture elements (*pixels*). Each pixel is just one dot of color. A bitmap is actually a *dot*mosaic, where the tiles are individual colored dots.

Instead of storing images as bitmaps, we usually store them using a compressed format such as a `GIF` or a `JPG`. Compressed images take up less memory on the disk or to download, but are harder to manipulate. Fortunately, the DrScheme environment includes functions for converting them to bitmaps so we don't need to worry about it. We can store our images in compressed formats and use library functions to convert them into bitmaps when we use them in our program.

The pixels in a bitmap are numbered on a grid. The top left corner is (0, 0). If the bitmap is w pixels wide and h pixels high, the bottom right corner is (w - 1, h - 1).

Selecting Tiles

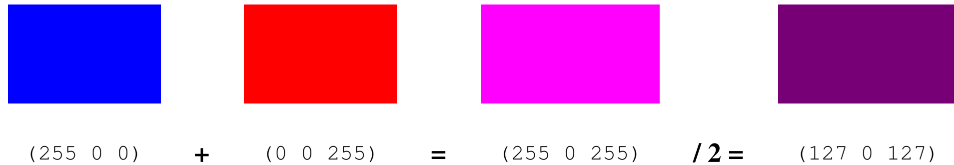
To select tile images for our photomosaic we follow steps 5 through 7 from the kindergartener description. We don't need to worry about programming step 6 (naptime), but if you are working with a partner do keep remembering to switch places with your partner.

To find a tile image that matches a rectangle of the master image, we need to know the colors of both the tile images and rectangles in the master image.

Finding the Average Color of a Bitmap

To find a matching tile image, we need to know the "color" of each tile. One way would be to calculate the average color of each pixel in the tile image by adding the red, green and blue values for each pixel and dividing by the number of

points. For example, if we wanted the average color of red `(255 0 0)` and blue `(0 0 255)` would sum the colors to get `(255 0 255)`, and divide by two to get `(127.5 0 127.5)`. This looks like purple, which is what you might expect:



(255 0 0) **+** (0 0 255) **=** (255 0 255) **/ 2 =** (127 0 127)

We can define a function that produces the sum of two colors by adding the red, green and blue components separately:

```
(define (add-color color1 color2)
  (make-color (+ (get-red color1) (get-red color2))
              (+ (get-green color1) (get-green color2))
              (+ (get-blue color1) (get-blue color2))))
```

To average many colors, we need to add all the colors together and then divide by the number of colors. One way to add a list of values together is to divide it into two simpler problems: add the first value to the sum of all the other values. That is,

```
SUM (a, b, c, d, e) = a + SUM (b, c, d, e)
```

We can do the same thing for the remaining values:

```
SUM (b, c, d, e) = b + SUM (c, d, e)
   SUM (c, d, e) =     c + SUM (d, e)
      SUM (d, e) =         d + SUM (e)
         SUM (e) =             e + SUM ()
          SUM () =                 0
```

The only tricky part is knowing what to do at the end when there are no colors left. We define the `SUM` of no values to be `0`. Similarly, when there are no more colors left, we evaluate to the `0`-color (black).

This is how we define `sum-colors`:

```
(define (sum-colors color-list)
  (if (null? color-list)         ;;; If there are no more colors in the list,
      (make-color 0 0 0)         ;;;    evaluate to the 0-color (black)
      (add-color                 ;;; Otherwise, evaluate to the result of adding
         (first color-list)      ;;;    the first color and
         (sum-colors             ;;;    the sum
           (rest color-list))))) ;;;        of all the other colors.
```

To calculate the average color of a bitmap, we just calculate the sum of all the colors of the pixels in the bitmap, and divide by the number of pixels.

For the photomosaic code, we don't want to look at every single pixel though since there are millions of pixels in a large image. Instead, we sample points from the image by selecting points at regular intervals. We do this by defining `generate-sample-points` a function that selects the points to sample, and applying `average-colors` to the result of `map`ping every sample point to the color of the bitmap point.

Dividing the Master Image

We need to divide the master image into tile-sized regions and calculate the average color of each region. This is almost the same as calculating the average color of a bitmap, except here we want the average color of lots of different regions within a single image.

Luckily we've already written a procedure that does almost what we need! We can use the `generate-sample-points` procedure to do this. To divide the master image, instead of just sampling the color of points, we generate a tile region with the sample point as its bottom left corner with the appropriate width and height.

Selecting the Tiles

We now know how to:

1. Load the tile images

2. Calculate the average color of each tile image
3. Divide the master image into tile size regions

Suppose we know how to find the best tile for one region. All we need to do make the whole photomosaic is do that for every region of the master image.

So, we define `select-mosaic-tiles` to take: a list of lists of color samples for regions in the master image (one list for each row) and a list of the tiles (where each tile is a list of a image name and the average color for that tile) and evaluate to a list of lists of images that describes the photomosaic.

Here's the rough code:

```
(define (select-mosaic-tiles samples tiles)
  (map2d find-best-match samples))
```

The `map2d` procedure (defined in *mosaic.ss*) is like `map`, except it works on lists of lists. It applies the procedure `find-best-match` to every inner element in a list of lists.

We need to pass the tiles to `find-best-match`, though, and we also want to pass a function for matching colors. So, the actual `select-mosaic-tiles` code is:

```
(define (select-mosaic-tiles samples tiles color-comparator)
  (map2d
   (lambda (sample)
     (tile-name (find-best-match sample tiles color-comparator)))
   samples))
```

The `find-best-match` function goes through the list of tiles, and selects the tile that best matches the sample color for this rectangle.

The `lambda` means "make a procedure", so the expression evaluates to a procedure that takes one parameter (`sample`) which is the average color of the master image for this square. It evaluates `(find-best-match sample tiles color-comparator)` to find the tile in `tiles` that best matches the `sample` color, using the `color-comparator` procedure to compare two colors (this is the procedure you will need to write for the last question).

Finding the best match is like summing a list of colors — we can break it down into checking if the first one is better than the best one of the rest of the list. That is, if we had a function `better-match` that can determine which one of two possible tiles is a better match for a particular sample, we can use it like `add-colors` to find the best match from a list of tiles. As with `add-colors`, we have to be careful about what happens when there are no colors to add. It doesn't make sense to find the best match if there are no tiles, so we produce an error in that case. If there is only one tile, we know that is the best one. Otherwise, we compare the first tile with the best one we found from the rest of the tiles.

Here's the code:

```
(define (find-best-match sample tiles color-comparator)
  (if (null? tiles)                 ;;; If there are no tiles,
      (error "No tiles to match!") ;;;    we lose.
      (if (= (length tiles) 1)     ;;; If there is just one tile,
          (first tiles)            ;;;    that tile is the best match.
          (pick-better-match       ;;; Otherwise, the best match is
           sample                  ;;; either the first tile in tiles,
           (first tiles)           ;;; or the best match we would find
           (find-best-match        ;;; from looking at the rest of the
              sample               ;;; tiles.  Use pick-better-match
              (rest tiles)         ;;; to determine which one is better.
              color-comparator)
           color-comparator))))

(define (pick-better-match sample tile1 tile2 color-comparator)
  (if (color-comparator sample (tile-color tile1) (tile-color tile2))
      tile1
      tile2))
```

Loading the Tiles and Master

We still need to figure out how to create the list of tiles. Suppose we start with a list of filenames of images and want to turn that into a list of bitmaps to use as the tiles. How can we break that into simpler problems?

If we have a function that can load one tile image, then all we need to do is apply that function to every filename in the list of image filenames.

We can do that with `(map get-one-image image-names)`. The function `map` applies a function (the first parameter) to every element in a list. We use this to define a function that takes a list of image filenames, and produces a list of the corresponding bitmaps:

```
(define (load-images image-names)
   (map get-one-image image-names))
```

Now, all we need are functions for getting a list of image filenames and for getting one image. We could just create the list of images by hand, but for a good photomosaic we need many images. So, we instead put all the tile images in a directory and use a function that gets a list of all files in a directory. DrScheme provides `directory-list` to do that. So, `(load-bitmaps (directory-list))` would evaluate to a list of bitmaps for every file in the current directory. The actual code provided is a bit more complicated since we want to handle directories that contain files that are not images and we want the name of the directory to be a parameter (instead of using the current directory).

Displaying the Photomosaic

One way to display the photomosaic is to create a web page with all the tile images in the right places. The result of `select-mosaic-tiles` is a two-dimensional list of all the tile images in the photomosaic.

For example, suppose the best tiling for a 3x3 photomosaic (for now, we are just using names of colors for the tiles) is:

red	red	red
white	white	white
blue	blue	blue

Then, `select-mosaic-tiles` would evaluate to:

```
(("red" "red" "red")
 ("white" "white" "white")
 ("blue" "blue" "blue"))
```

We can display this using the divide-and-conquer approach: display the first row of images at the top, and then display the rest of the images below it. Displaying one row of images is similar: put the first image at the left of the screen, and then put the rest of the images to its right.

Our rough code is:

```
(define (display-tiles output-file tiles)
  (map display-one-row tiles))
```

We define rough code for `display-one-row` similarly:

```
(define (display-one-row output-file row)
    (map display-one-tile row))
```

The actual code is a little more complicated since we need to pass the `output-file` and to produce good photomosaics we also need some additional parameters (like the size of the tiles).

To produce web pages, we need to put some formatting codes before and after the tiles. (You shouldn't need to look at these codes.) So, we define procedures that print the necessary codes before and after the tiles (`print-page-header` and `print-page-footer`). The `produce-tiles-page` procedure first prints the page header, then displays all the tiles, and finally prints the page footer.

Putting Everything Together

We need to combine all the steps now into one procedure that produces a photomosaic. Here is it:

```
(define (make-photomosaic
         master-image               ;;; Filename for the "big" picture
         tiles-directory            ;;; Directory containing the tile images
```

```
        tile-width tile-height     ;;; Display width and height of the tiles
                                   ;;;    (doesn't have to match actual size)
        sample-width sample-height ;;; Sample width and height (each tile covers this size
                                   ;;;    area in master)
        output-filename            ;;; Name of file to generate (.html)
        color-comparator)          ;;; Function for comparing colors
  (produce-tiles-page
   output-filename
   (choose-tiles (get-one-image master-image)
                 (load-tiles tiles-directory)
                 sample-width sample-height
                 color-comparator)
   tile-width tile-height))
```

We take parameters for the master image, directory containing the tiles, tile width and height, sample width and height, the output filename to generate, and the function to compare colors (which you will write).

We then call `produce-tiles-page` with the output filename, the mosaic tiles (calculated using `choose-tiles`, and the tile width and height parameters.

Matching Colors

We're all done now, except for the procedure that determines which of two colors is a better match for another color. That is left for you to do.

Question 5: Write a function that can be passed as `color-comparator`. Define your function in the definitions file you created for Question 4. Use your function to create a photomosaic web page.

Your function should look like,

```
(define (closer-color?  ;;; The name of your function.
        sample          ;;; The average color of the rectangle you want to match.
        color1          ;;; The average color of the first tile.
        color2)         ;;; The average color of the second tile.
   ;;; A Scheme expression that evaluates to #t if color1 is a better
   ;;; match for sample, and #f otherwise.
   )
```

In addition to the standard arithmetic operators (`+`, `*`, `-`, `/`), comparison operators (`<`, `>`, `=`), provided procedures for manipulating colors (`get-red`, `get-green` and `get-blue`), some other procedures you may find useful include:

- `(abs` *`number`*`)` — evaluates to the absolute value of number parameter. For example, `(abs -3) ==> 3`.
- `(square` *`number`*`)` — (defined in *mosaic.ss*) evaluates to the square of the number parameter. For example, `(square 2) ==> 4`.
- `(sqrt` *`number`*`)` — evaluates to an approximation of the square root of the number parameter. For example, `(sqrt 2) ==> 1.4142135623730951`.

You should test out your `closer-color?` procedure using some simple colors:

```
> (closer-color? white white black) ;; white is closer to white than black is
#t
> (closer-color? white black white) ;; black is not closer to white than white is
#f
> (closer-color? red magenta red) ;; magenta is not closer to red than red is
#f
> (closer-color? red orange blue) ;; orange is closer to red than blue is
#t
```

Once your `closer-color?` procedure works for the simple examples, you can try to create a photomosaic using ot by evaluating:

```
(make-rotundasaic output-filename closer-color?)
```

The value passed as *output-filename* should be the full pathname of a file you can create that does not already exist, for example to generate output in the file `J:/cs150/ps1/mosaic.html` you would evaluate

```
(make-rotundasaic "J:\\cs150\\ps1\\mosaic.html" closer-color?)
```

If your `closer-color?` procedure is good, you should be able to recognize the rotunda when you open the `output.html` file in a web browser. Experiment with different ways of deciding which two colors are closest to improve your photomosaic, but try simple things first. A good `closer-color?` procedure should produce a photomosaic similar to this one: *sample.html*.

Teasers

Mosaics touch on many computer science issues, some of which we will return to later in the course.

Higher Order Functions. We used `map` to apply a function to every element of a list. `map` is an example of a *higher order function* — that means it is a function that takes other functions as parameters. Higher order functions can be very powerful. In Problem Set 2 (and throughout the course), we will explore higher order functions and recursion.

Lists. Many functions in this assignment operated on *lists*. In Problem Set 2, you will understand lists and implement functions that operate on lists (for example, you will be able to implement `map`).

Computational Complexity. Our program might use the same tile image over and over again. If there is a large section of the same color in our master image, the same tile image will be repeated for that whole section. This looks pretty bad. A real mosaic would never reuse the same tile. Producing a non-duplicating photomosaic is an extremely hard problem. In fact, it is such a hard problem that no one knows for sure whether or not there is a *fast* solution (we'll explain exactly what we mean by fast here later in the course) to finding the best photomosaic tiling. Computer scientists call this an *NP-hard* problem. If someone discovers a fast way to produce the best possible non-repeating photomosaic, then it means there are also fast solutions to many other seemingly unrelated and important problems (such as the "travelling salesperson problem" where the goal is to find the best route for a salesperson to visit a set of cities, and the "cure most deadly diseases problem" where the goal is to find a sequence of protiens that folds into a particular shape). We will explore this about halfway through the course.

Searching. Our program takes a long time to run for even small photomosaics. A really smart kindergartner wouldn't find a good match by looking through all the images scattered haphazardly on the floor; instead, she would sort the images into groups based on their color. Then she could easily see all the bluish images together, and pick the one that best matches a particular square. Our program has to consider every image for every rectangle. A better program would be able to quickly find the best image without having to consider every image every time. Computer scientists do a lot of work on finding better ways to search. Google's database has 8,168,684,336 web pages (as of 21 August 2005, they don't publicize their database size any more). If it searched them the same way we search for images and could consider 10000 pages per second, it would take over three days to get a response to your query:

```
> (exact->inexact ;;; try evaluating it without this to see why it is here
   (/ (/ 8168684336 10000) (* 60 60 24)))
9.45449575925926
```

We will look at some better ways of searching in a few weeks.

Concurrency. One thing to note about our problem division solution, is that some of the steps can be done in different orders, but others can only be done after an earlier step is already complete. For example, we can do steps 3, 4 and 5 before step 1, but we cannot do step 7 before step 1. If we have a whole class of kindergartners to help make our photomosaic, we could give each of them a magazine and have them all collect tile pictures at the same time. One way to make computers solve a problem quicker is to divide the problem into pieces that can be done at the same time, and have different computers work on each piece at the same time.

Photomosaic-mosaics. A photomosaic is a mosaic made with photo tiles. We could also make a *photomosaic-mosaic*, a mosaic made with photomosaic tiles. Of course, why stop there? We could make a *photomosaic-mosaic-mosaic*, a mosaic where the tiles are photomosaic-mosaics. If we have a lot of computing power, we could make a *photomosaic-movie*, a movie where the frames are photomosaics, or a *moviemosaic-photo*, a still image where the tiles are movies instead of still images. There is one *moviemosaic-movie* (or perhaps more accurately, *moviemosaic-mosaic-mosaic-movie*) that has been seen by several billion people! Identifying it is worth a token reward (hint: it was eight and a half years ago).

Software Patents. Robert Silvers claims to have a patent (United States Patent 6,137,498: Digital composition of a mosaic image) on Photomosaics. We're not sure what the ancient Babylonians have to say about this, but software patents raise lots of complex legal issues. One would have to check with a lawyer to determine if you can redistribute your code from this problem set, but we encourage you to do so and would be happy to defend a test case against this patent.