Brandon Beckwith 800863439
Haely Pratt 800876674
Bryson Roland 800862143

Term Project: Part 2
Common Programs

Find the Missing Element

For any array of non-negative integers, and a second array formed by shuffling the elements of the first array and removing a random element, the removed element is found by finding the difference in the sums of each array. That is, if A is the first array with elements $a_1, a_2, \dots, a_n$, and B is the second array with elements $b_1, b_2, \dots, b_{n-1}$, then the missing element x is found by:

$$x = \left[\sum_{i=1}^{n} a_i\right] - \left[\sum_{i=1}^{n-1} b_i\right]$$

Compared with a linear search, which in the worst case would require $n(n-1)$ array accesses and additional memory for storing both arrays, this solution reduces the problem to $n + (n-1)$ array accesses and only requires memory for storing one array and its sum. In Perl, this is achieved by the following code:

```
# Input array stored in @nums
# Print out the array
print "@nums\n";

# Sum the array
my $sum1 = eval join '+', @nums;

# Shuffle the array using a built-in subroutine
@nums = shuffle @nums;

# Delete a random element from the array
splice @nums, rand(scalar @nums), 1;

# Print the modified array
print @nums == 0 ? "[]\n" : "@nums\n";

# Calculate the missing element (old – new) = missing
print "Missing: ".($sum1 – (@nums == 0 ? 0 : (eval join '+', @nums)))."\n";
```

Here, a default array [0, 1, 2, 3, … , 15, 16] is provided when the user does not pass any arguments to the function, and the case where the user passes a single element as an argument is also handled. Note that we have removed code headers, import statements, and initialization steps in this code block and all blocks following to conserve space in this document.

Check Balanced Parentheses

Given a string of parenthetic characters—round brackets: **()**, square brackets **[]**, and curly braces: **{}**—containing no other characters, one can determine if the parentheses are balanced by iteratively removing consecutive closed pairs until either no characters remain (i.e. balanced condition), or until no more consecutive closed pairs of parentheses can be found (i.e. unbalanced condition). Matches are found using regular expressions. In Perl, this is achieved by the following code:

```
# Input stored in $_
# Loop and remove each matched pair
while(1){
  # If we can match a pair, remove it
  if (s/(\{\}|\[\]|\(\))//xs) {
    # If the string is empty, it's balanced
    if ($_ eq ''){
      print "Balanced!\n";
      last;
    }
  } else {
    # If we can't match, then the string is unbalanced
    print "Unbalanced\n";
    last;
  }
}
```

Permutations

To generate all permutations of a given string, we use the iterative version of Heap's algorithm. This method systematically swaps elements in the array of characters, generating all permutations of the first $(n-1)$ elements with a fixed last element. Once all permutations of the first $(n-1)$ elements are found, the first element and last element are swapped, and the process repeats until all permutations have been generated.

For example, for a string of four characters, we first find all permutations of the first two characters with the last two characters fixed, then swap the first element with the third element, again find all permutations of the first two characters with the last two characters fixed, swap the first and third element again, find all permutations of the first two characters, then we can swap the first and last elements and repeat the entire process all over again. In Perl, this is achieved by the following code:

```
# Input characters stored in array @chars of length $len
# Heap's algorithm
my $i = 0;
my @j = (0) x $len;

print "@chars\n";

while ($i < $len){
  if ($j[$i] < $i){
    if(($i % 2) == 0){
      @chars[0,$i] = @chars[$i,0];
    } else {
      @chars[$j[$i],$i] = @chars[$i,$j[$i]];
    }
    print "@chars\n";
    $j[$i]++;
    $i = 0;
  } else {
    $j[$i] = 0;
    $i++;
  }
}
```

Note that this code differs slightly from the submitted version: namely, we print to the terminal in a format that is equivalent to the example in the assignment instructions.

File I/O

To count the number of words in a text file, we iterate over each line of the text file provided as an argument and use regular expressions to match groups of word characters. For each line, we add the number of word-character group matches to an accumulator until the end of the file has been reached. We then create a second text file (named by "out_" prepended to the name of the input text file) and print the value of the accumulator to it. In Perl, this is achieved by:

```
# The name of the input text file is stored in $name
# Loop through all lines in the file given as an argument
while (my $line = <>){
  # Use regex to count the words in a line and add them up
  $count += () = $line =~ /\w+/g;
}

# Open a file, write the word count, and close the file
open (my $fout, ">", "out_$name") or die "Couldn't open out_$name"\n";
print $fout $count;
close $fout;
```

Note that the diamond operator (**<>**) in Perl is used to automatically read from the file(s) specified in the command line arguments. It does not require any special code to open, or read from, the specified file.

Three-of-a-Crime

Firstly, as Perl does not natively support GUIs, we use the popular Tk module provided in CPAN. This is easily installed using the Perl Package Manager (PPM) by running **`ppm install Tk`** in a terminal window. Since most of the code provided in **`ThreeOfACrime.pl`** does not relate to the game algorithm and is cumbersome to provide in a document, we include only relevant snippets here.

In linking the logic of the game to a GUI, we organize the game states into "pages," which are all initialized at the start of the script. This way, we can transition to a new game state without tearing down GUI elements and subsequently recreate them (with the exception of Player labels). Instead, we hide members of a page and simply show members of the page we are transitioning to.

Before the start of the game, a "start page" is displayed, allowing users to choose a 1-, 2-, or 3-player game, or to quit.

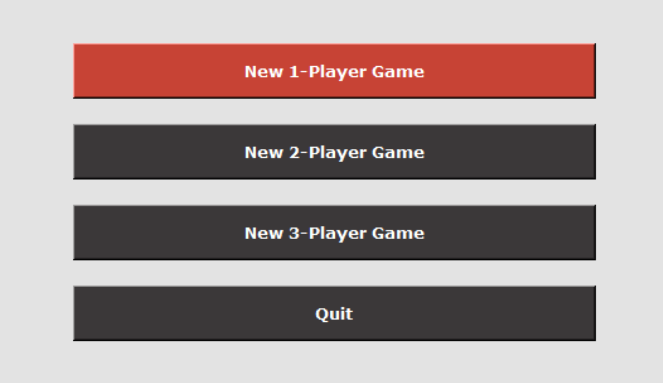

Figure 1. Start Page

Once a game is started, players are initialized in a hash containing each player's current state (active, inactive, or disabled) and their corresponding GUI label; three randomly selected criminals are chosen as

perpetrators, and three randomly selected criminals are chosen for display. The game starts with Player 1, and the page transitions to the "player turn" page, where the currently player is highlighted in green, a dialog displays how many of the displayed criminals are perpetrators, and player turn buttons allow the player to either identify the criminals or pass to the next player.

Figure 2. Player Turn Page

```
# Initializing the perpetrators and displayed criminals in Perl. Note that we
# use hard-coded values here, but not in the submitted code

sub init_perpetrators{
  # Selects a random subset of 3 criminals from the set of 1 thru 7
  @PERPS = (shuffle 1..7)[0..2];
}

sub init_criminals{
  do{
    # Select a random subset of criminals such that no more than 2 of them
    # are in the set of perpetrators
    @CRIMS = (shuffle 1..7)[0..2];
    $PERPS_DISP = intersection(\@CRIMS, \@PERPS);
  }while($PERPS_DISP > 2);
}

sub intersection{
  # Get the intersection of two lists
  my $a = shift; my $b = shift; my @isect = ();
  foreach my $e (@{$a}){ push(@isect, $e) if $e ~~ @{$b}; }
  return @isect;
}
```

In initializing the random subset of criminals to display, we determine how many of these are in the set of perpetrators by counting the number of elements in the intersection of the two lists. This logic is also used

to determine if a player has identified the correct criminals as perpetrators. If a player chooses to identify the perpetrators, the "criminal selection" page is displayed. Here, all criminals are displayed, and the player can choose exactly three of these as the perpetrators. That is, the "Done" button is disabled until three criminals are selected, and all unchecked criminals will be disabled when three criminals are selected.

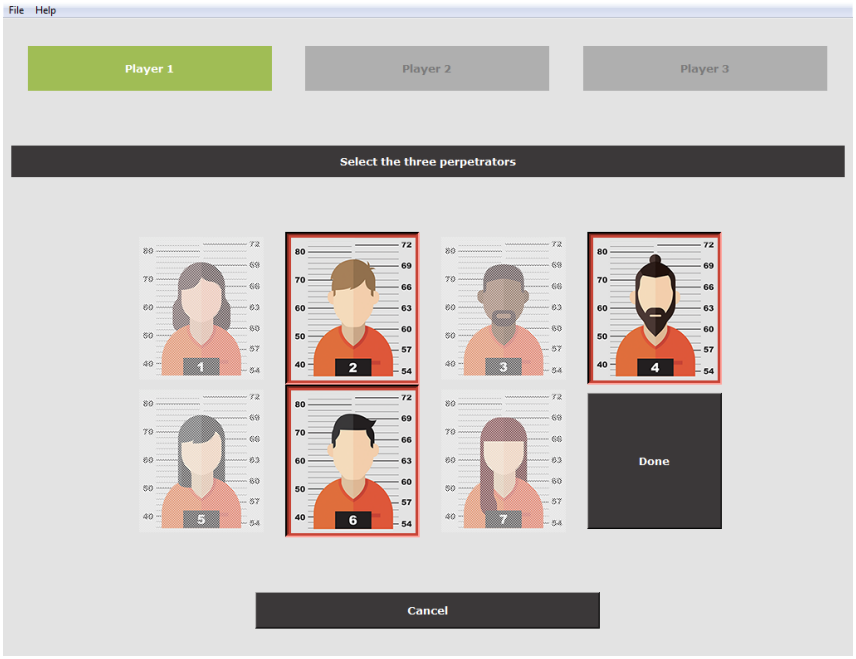

Figure 3. Criminal Selection Page

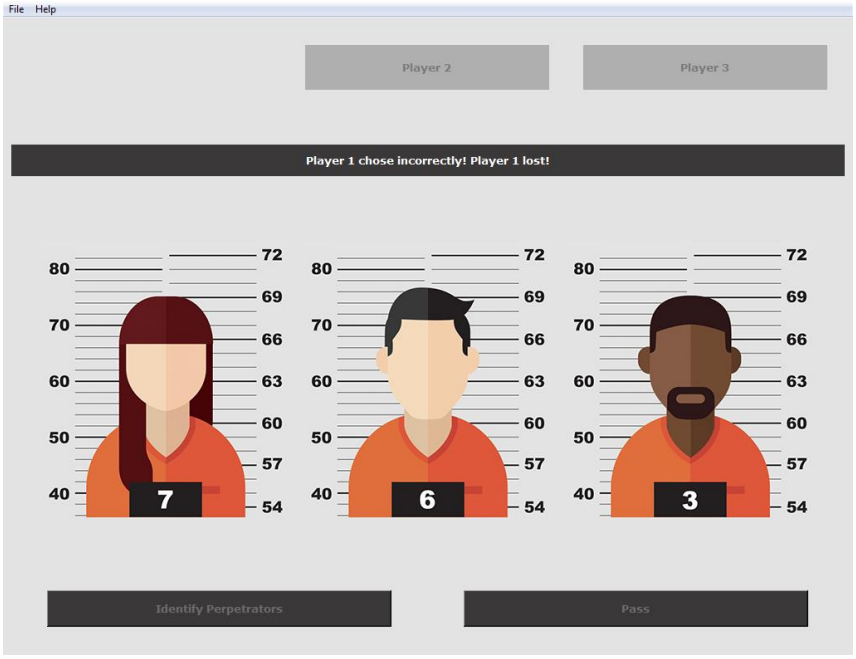

Figure 4. Player Lost

When an incorrect selection is submitted, the player's state is set to "disabled" and the turn is passed to the next player. Once each player has taken a turn (at least one by passing), a new subset of criminals is selected and displayed. The pass subroutine is found below:

```
sub pass{
  if($REM_PLAYERS == 0){
    # If no players remain, the game has been lost,
    # so transition to the 'Game Lost' page
    set_page(&PG_GMLST);
  }else{
    # Set the current player to inactive unless they have lost the game
    unless($PLAYERS{$CURR_PLAYER}{'state'} eq &DISABLED){
      set_player_state($CURR_PLAYER, &INACTIVE);
    }
    # Move to the next available player; if the end of the list of players has
    # been reached, display a new set of criminals and the respective dialog,
    # and wrap the player turn marker
    do{
      $CURR_PLAYER++;
      if($CURR_PLAYER > keys %PLAYERS){
        $CURR_PLAYER = 1;
        init_criminals();
      }
    }while($PLAYERS{$CURR_PLAYER}{'state'} eq &DISABLED);
    set_player_state($CURR_PLAYER, &ACTIVE);
  }
}
```

If all players choose incorrectly, the game ends and moves to the "game over", or "game lost", page. Otherwise, if a player chooses correctly, the game ends and moves to the "game won" page. Details about page transitions, GUI elements, and general game states are found in the internal documentation of the corresponding Perl script.

Figure 5. Game Lost Page

Figure 6. Game Won Page