cs150: Exam 1 - Comments

Due: Monday, 26 February at 12:02pm

Name: ***Kaume Mentz***

Directions

Work alone. You may not discuss these problems or anything related to the material covered by this exam with anyone except for the course staff between receiving this exam and class Monday.

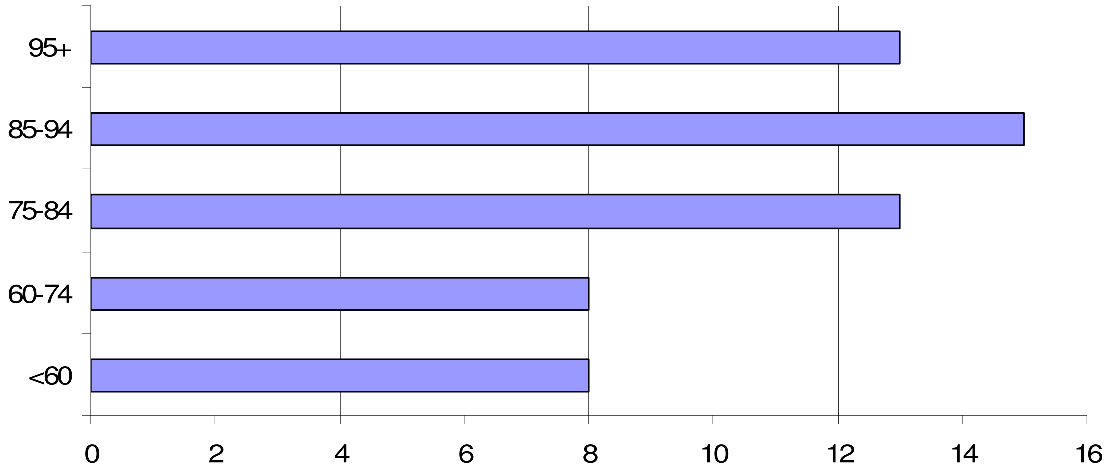
Open resources. You may use any books you want, lecture notes, slides, your notes, and problem sets. You may also use DrScheme, but it is not necessary to do this. You may also use external non-human sources including books and web sites. If you use anything other than the course books, slides, and notes, cite what you used. You may not obtain any help from other humans other than the course staff.

Answer well. Answer all questions 1-9 (question 0 is your name, which hopefully everyone will receive full credit for), and optionally answer questions 10-12. The questions are not necessarily in order of increasing difficulty, so if you get stuck on one question you should continue on to the next question. There is no time limit on this exam, but it should not take a well-prepared student more than a few hours to complete. It may take you longer, though, so please do not delay starting the exam. There is no valid excuse (other than a medical or personal emergency) for running out of time on this exam. (Abridged from original exam.)

Full credit depends on the clarity and elegance of your answer, not just correctness. Your answers should be as short and simple as possible, but not simpler. Your programs will be judged for correctness, clarity and elegance, but you will not lose points for trivial errors (such as missing a closing parenthesis).

Average Scores

0	1	2	3	4	5	6	7	8	9	Total
9.95	6.37	4.86	7.54	9.22	8.18	8.46	8.60	7.82	7.96	79.0



1. The and-expression is a special form for logical conjunction. An and-expression has any number of operand expressions. For example, all of the expressions below are valid and-expressions (question 2 will describe the evaluation rule for and-expressions):

```
> (and)
#t
>(and (= 2 2) (= 3 3))
#t
>(and 3 6 2 5 3 #f 2 5 (car 3))
#f
```

Define a BNF grammar rule for the *AndExpression*. You may assume all the grammar rules from Chapter 3 are defined and use them in your answer.

Easiest answer:
AndExpression ::= **(and** *MoreExpressions***)**

If you were worried that was too easy (which apparently many people were!), you could write it without using *MoreExpressions*:

AndExpression ::= **(and** *ZeroOrMoreExpressions***)**
ZeroOrMoreExpressions ::= ε
ZeroOrMoreExpressions ::= *Expressions ZeroOrMoreExpressions*

2. For simplicity, the rest of this question assumes a limited version of the and expression that only takes two operands: (and $expr_1$ $expr_2$) This simplified and-expression behaves identically to the standard and-expression when applied to two operands, but is not defined for other than two operands. The evaluation rule for an and-expression is:

> To evaluate an and-expression, evaluate the first subexpression. If it evaluates to a false value, the value of the and-expression is false. Otherwise, the value of the and-expression is the value of the second subexpression.

Dana Carver doesn't like unnecessary special forms and suggest that the and special form can be replaced with this procedure:

```
(define (and e1 e2) (if e1 e2 #f))
```

Provide a convincing argument that the `and` procedure above is not equivalent to the and-expression special form. (Hint: describe inputs where they mean different things.)

The key different is that the special form and only evaluates the second operand expression if the first one evaluates to a true value, whereas any procedure application must evaluate all subexpressions. This different is apparent if the second expression does not evaluate to a value (either because it never finishes evaluating or because it there is an evaluation error). For example,

```
(and #f (+ #f 7))
```

evaluates to `#f` using the special form and expression, but would produce and error when the `+` application is evaluated with the procedure.

For convenience, here is the find-maximum procedure from Chapter 4:

```
(define (find-maximum f low high)
    (if (= low high)
        (f low)
        (max (f low)
             (find-maximum f (+ low 1) high)))))
```

3. (This is a slight rewording of Question 2 of notes 5 and Exercise 4.8 in the book.) The `find-maximum` procedure we defined in Chapter 4 (and Notes 5) evaluates to the maximum value of the input function in the range, but does not provide the input value that produces that maximum output value. Define a procedure, `find-maximizing-input` that takes the same inputs as `find-maximum`, but outputs the input value in the range that produces the maximum output value.

For example:

```
> (find-maximizing-input (lambda (x) x) 3 150)
150
> (find-maximizing-input (lambda (x) (- (* 12 x) (* x x))) 0 50)
6
```

For maximum credit, your answer should have running time in $\Theta(n)$ where n is the difference between the values of the `high` and `low` inputs. (But you will receive most of the credit even if your solution is less efficient.)

The simplest definition is probably:

```
(define (find-maximizing-input f low high)
   (if (= low high)
       low
       (if (> (f low)
              (f (find-maximizing-input f (+ low 1) high)))
           low
           (find-maximizing-input f (+ low 1) high))))
```

The problem with this definition is each application involves **two** recursive calls. This means the growth will be similar to the slow Fibonacci procedure! It is in $\Theta(2^n)$ where n is the difference between the values of the high and low inputs since increasing the different by one *doubles* the amount of work to do.

To obtain a linear-time procedure, we need to avoid evaluating the recursive application twice. The easiest way is to use a let expression:

```
(define (find-maximizing-input f low high)
   (if (= low high)
       low
       (let ((best (find-maximizing-input f (+ low 1)
                                          high)))
          (if (> (f low) (f best))
              low
              best))))
```

4. Define a `make-incrementer` procedure that takes one input, the increment number, as input and produces as output a procedure. The output procedure is a procedure that takes one number as input, and produces as output the value of that number increased by the increment number.

For example:

```
> (make-incrementer 1)
#<procedure>
> ((make-incrementer 2) 148)
150
>((make-incrementer 3) ((make-incrementer 7) 1))
11
```

```
(define (make-incrementer n)
   (lambda (x) (+ x n)))
```

5. Define a procedure `find-worst` that takes two inputs: a list and a comparison procedure. As output it produces the element in the list which is the worst according to the comparison procedure.

For example:

```
> (find-worst (list 1 5 0) <)
5
```

There are many possibilities. One is similar to the way we defined find-best:

```
(define (find-worst lst cf)
    (if (null? (cdr lst)) (car lst)
        (if (cf (car lst) (find-worst (cdr lst) cf))
            (find-worst (cdr lst) cf)
            (car lst))))
```

Another possibility (which is very inefficient) is to use sort and reverse to find the last element:

```
(define (find-worst lst cf)
  (car (reverse (sort lst cf))))
```

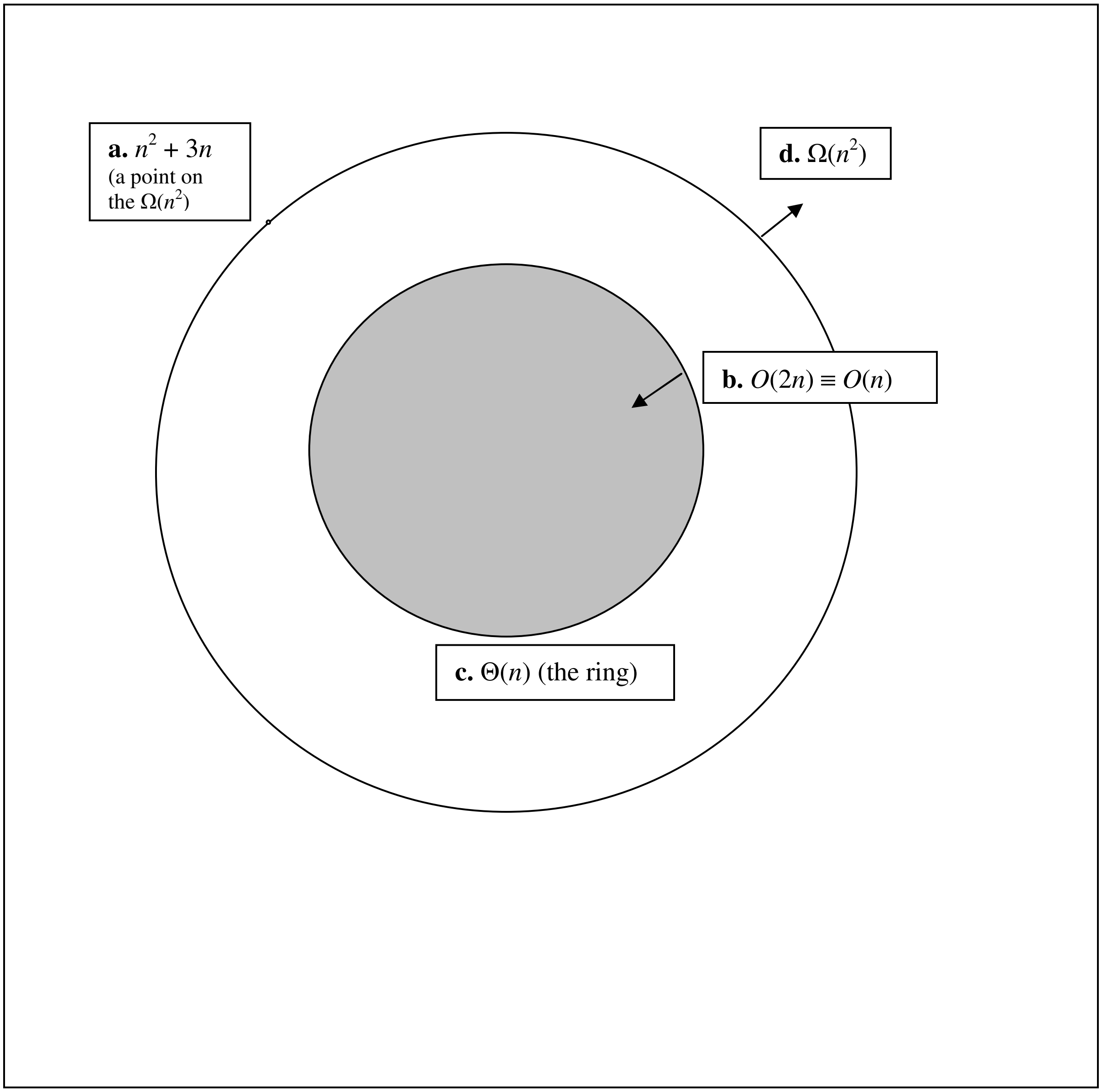
We could also define it using find-best, but swapping the comparison inputs:

```
(define (find-worst lst cf)
  (find-best lst (lambda (a b) (cf b a))))
```

6. Draw a picture illustrating the asymptotic growth rates of the following functions and sets of functions:

a. $n^2 + 3n$
b. $O(2n)$
c. $\Theta(n)$
d. $\Omega(n^2)$

The center of your picture should be the slowest growing functions, and as you move further from the center, functions grow faster (similar to Figure 6.2 of the book). If you are depicting a set, use arrows or color to make it clear what space is included in the set. (There is no need for a fancy drawing. It is fine to hand draw something clear.)


a. $n^2 + 3n$
(a point on the $\Omega(n^2)$
d. $\Omega(n^2)$
b. $O(2n) \equiv O(n)$
c. $\Theta(n)$ (the ring)

Another cryptographic invention of Alan Turing's at Bletchley Park was a process then known as *banburismus* (more commonly now called *sequential analysis*) developed to determine if two intercepted Enigma messages had been encrypted using the same key.

The goal of the banburismus technique is to determine when two intercepted Enigma messages were encrypted using the same or similar initial machine settings. The key insight is identical to that behind the double delta technique used by Colossus: since the letter distribution in a natural language (in this cases German) is not even, it is much more likely that the same letters will occur at a given position than would occur by random chance (which is approximately the case if the Enigma machines were not configured with similar wheel settings).

So, to determine if two messages were sent by Enigma machines with the same wheel settings we need to count the number of occurrences in the ciphertext where the two messages have the same letter at the same position. If that number significantly exceeds the number predicted by random chance, then it is likely the messages were encoded using the same wheel settings.

7. Define a procedure `count-matches` that takes as input two lists of characters (representing two intercepted ciphertexts) and outputs a number that is the number of positions where the characters in the two lists match (use `eq?` to test two characters for equality).

For example,

```
> (count-matches (list #\A #\B #\C) (list #\B #\B #\C)
2
> (count-matches (list #\A #\B #\C) (list #\A))
1
>(count-matches (list #\A #\B #\C) (list #\A #\B #\C #\D))
3
```

```
(define (count-matches lst1 lst2)
  (if (or (null? lst1) (null? lst2))
      0
      (+ (if (eq? (car lst1) (car lst2)) 1 0)
         (count-matches (cdr lst1) (cdr lst2)))))
```

Since the Enigma wheels rotate, it was also possible to use this technique to find messages sent using similar initial configurations by trying different alignments of the two messages. For example, suppose the intercepted messages are:

```
Message 1: GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
Message 2: YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

(this example is from http://en.wikipedia.org/wiki/Banburismus)

The Bletchley Park analysts would try aligning the messages at different starting points, looking for ways of aligning them that have a high number of matches. For example:

Align 0:

```
GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

Align +1

```
GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
 YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

...

Align +9

```
GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
         YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

With the Align +9, there are 9 matches which is promising (that is, it would be very unlikely to occur by chance, so the wheel settings are probably similar).

8. Define a procedure `find-best-alignment` that takes as input two lists, representing two intercepted ciphertexts, and outputs the number of matching letters in the best possible alignment.

A good answer will find the best positive alignment (only considering moving the second message to the right, as in the example above). An excellent answer will consider both positive and negative alignments (moving the second message to the left instead).
For example:

```
> (find-best-alignment (list #\A #\B #\C) (list #\B #\B #\C))
2
> (find-best-alignment (list #\A #\B #\C #\D) (list #\B #\C #\D))
3
> (find-best-alignment (list #\A #\B #\C #\D)
                       (list #\F #\A #\B #\C #\D))
4
```

This is the correct result for "excellent" answers that consider negative alignments. The answer using just positive alignments is `0`.

Hint: note that the letters with no corresponding letter in the other message don't matter. So, we could view Align +1 above as

```
XCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

without the leading `G` in message one.

(Answer space is on the next page)

8 (continued). Define your `find-best-alignment` procedure here:

The first procedure is a "good" solution, it finds the best alignment value, only trying positive alignments:

```
(define (find-best-pos-alignment msg1 msg2)
  (if (or (null? msg1) (null? msg2))
      0
      (pick-better
       (count-matches msg1 msg2)
       (find-best-pos-alignment (cdr msg1) msg2)
       >)))
```

To find the best alignment considering both directions, we need to slide both left and right. The easiest way to do this is to just switch the order of the operands:

```
(define (find-best-alignment msg1 msg2)
  (pick-better
   (find-best-pos-alignment msg1 msg2)
   (find-best-pos-alignment msg2 msg1)
   >))
```

9. Analyze the running time of your `find-best-alignment` procedure. Your analysis should include a description of the running time using Θ notation.

Our procedure has running time in $\Theta(n^2)$ where n is the total number of elements in the input lists.

The `find-best-pos-alignment` procedure cdr's down the first input list. It will make n_1 recursive calls, where n_1 is the number of elements in the first input list. Each application involves an application of the constant time procedures null? and pick-better (from class), as well as the count-matches procedure we defined for question 7. That procedure cdrs down the input lists, doing constant work each time, so it has running time in $\Theta(n)$ where n is the length of the shortest input list. In the worst case, msg1 and msg2 are the same lengths initially, so on average, this is ½ the length of the first input to find-best-pos-alignment, which is $\frac{1}{2} n_1$. So, the total running time for the find-best-pos-alignment application is in $\Theta(n^2)$: there are $\Theta(n)$ applications of a procedure that has running time in $\Theta(n)$.

The find-best-alignment procedure applies find-best-pos-alignment twice, so its running time is in $\Theta(n^2) + \Theta(n^2)$ which is equivalent to being in $\Theta(n^2)$.