# ECE368 Exam 1
# Spring 2014

*Wednesday, February 26, 2014*
*6:30-7:30pm*
*PHYS 112*

*READ THIS BEFORE YOU BEGIN*

This is an *open-book, open-notes* exam. Electronic devices are not allowed. The time allotted for this exam is *exactly* 60 minutes. *It is in your best interest to not spend too much time on any problem.*

*Always show as much of your work as practical*–partial credit is largely a function of the *clarity and quality* of the work shown. Be *concise*. It is fine to use the blank page opposite each question (or at the back of each question) for your work. Do draw an arrow to indicate that if you do so.

This exam consists of 7 pages; please check to make sure that *all* of these pages are present *before* you begin. Credit will not be awarded for pages that are missing – it is *your responsibility* to make sure that you have a complete copy of the exam.

Learning objective 1 can be satisfied if you score 10 points or more in question 2 or question 3. Learning objective 2 can be met if you obtain 10 points or more in question 1 or question 2. Learning objective 3 can be achieved by scoring 10 points or more in question 1.

**IMPORTANT:** *Read and sign* the *Academic Honesty Statement* that follows:

*"In signing this statement, I hereby certify that the work on this exam is my own, that I have not copied the work of any other student while completing it, that I have not obtained help from any other student, that I will not provide help to any other student, and that I will not disclose the contents of this exam until I receive the graded exam from the instructor. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action."*

`Printed Name:`

`login:`

`Signature:`

## DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

**1. Shell Sort (20 points)** Consider the following procedure that performs Shell sort to sort `n` integers in the array `r[0..n-1]`. The integers in the sequence used for Shell sorting are assigned to the variable `k`.

```
Shell_Sort(r[0..n-1])
01.     k ← 1;
02.     p ← 1;
03.     while (k < n) {
04.         k ← k × 3 + 1;
05.         p ← p + 1;
06.     }                              /* end of while (k < n) */
07.     k ← (k - 1)/3;
08.     p ← p - 1;
09.     while (k >= 1) {
10.         for j ← k to (n - 1) { /* insertion sort of k subarrays */
11.             temp_r ← r[j];
12.             i ← j;
13.             while (i >= k and r[i-k] > temp_r) {
14.                 r[i] ← r[i-k];
15.                 i ← i-k;
16.             }                      /* end of while (i >= k and ...  */
17.             r[i] ← temp_r;
18.         }                          /* end of for j ← k to (n - 1) */
19.         k ← (k - 1)/3;
20.         p ← p - 1;
21.     }                              /* end of while (k >= 1) */
```

**(a) (4 points)** What are the integers assigned to `k`'s in lines 01–06 when arrays of the following sizes (`n`) are passed into the function? Write down each sequence of integers assigned to `k` in the order in which they are assigned.

- n = 1:
- n = 4:
- n = 10:
- n = 30:

**(b) (9 points)** There is a closed-form formula for the integers assigned to `k`'s in lines 01–06:

$$\mathtt{k} = \frac{3^{\mathtt{p}} - 1}{2}.$$

For lines 03, 09, and 19, write down the expression for the number of times each of the instructions is executed in terms of `n`. You may have to use the ceiling or the floor function. Given a real number, $x$, the ceiling of $x$, denoted as $\lceil x \rceil$ is the smallest integer that is not less than $x$. The floor of $x$, denoted as $\lfloor x \rfloor$ is the largest integer that is not greater than $x$. In other words, $\lfloor x \rfloor \leq x \leq \lceil x \rceil$.

Line 03:

Line 09:

Line 19:

**(c) (7 points)** Consider an array of 23 integers, `r = [22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0, 21, 19, 17, 15, 13, 11, 9, 7, 5, 3, 1]`. The first `k` used for Shell sorting is 13. Show the array `r` right after 13-sorting.

**2. Recursion and Iteration (20 points)** Consider the function $f(x) = x^n$, where $x > 0$ is a real number and $n \geq 0$ is a non-negative integer. A recursive definition of $f(x) = x^n$ is given below:

$$f(x) = x^n = \begin{cases} 1 & \text{if } n = 0; \\ x & \text{if } n = 1; \\ x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} \times x & \text{if } n > 1 \text{ and } n \text{ is odd;} \\ x^{n/2} \times x^{n/2} & \text{if } n > 1 \text{ and } n \text{ is even.} \end{cases}$$

**(a) (8 points)** Consider the following recursive implementation developed by Prof. Koh:

```
Rec_Power(x, n)
     if (n == 0)
         return 1.0;
     if (n == 1)
         return x;
     if (n  mod  2 == 1) /* n is odd */
         return Rec_Power(x, n/2) * Rec_Power(x, n/2) * x; /* integer division */
     else                /* n is even */
         return Rec_Power(x, n/2) * Rec_Power(x, n/2);     /* integer division */
```

**(i) (4 points)** Draw the computation tree that shows the recursive calls when we call `Rec_Power(3.0, 10)`, i.e., $x = 3.0$ and $n = 10$.

**(ii) (4 points)** What is the time complexity of Prof. Koh's recursive implementation in terms of $n$? What is the space complexity of Prof. Koh's recursive implementation in terms of $n$?

**(b) (4 points)** You believe that Prof. Koh's recursive implementation is inefficient and that you can have a recursive implementation that has a time complexity of $O(\log n)$. Write down your $O(\log n)$ recursive implementation `Fast_Rec_Power` below:

```
Fast_Rec_Power(x, n)
     if (n == 0)
         return 1.0;
     if (n == 1)
         return x;
     /* complete your implementation in the space below */
```

**(c) (8 points)** Consider the following iterative implementation:

```
Itr_Power(x, n)
     pdt ← 1.0;
     tmp ← x;
     while (n > 0) {
         if (n  mod 2 == 1) /* n is odd */
             pdt ← pdt * tmp;
         tmp ← tmp * tmp;
         n ← n/2; /* integer division */
     }
     return pdt;
```

**(i) (4 points)** In the following table, write down the values of `pdt`, `tmp`, and `n` as they are updated in the `while`-loop when we call `Itr_Power(3.0, 10)`. You may write down the values of `pdt` and `tmp` in the form of $3^i$, where $i$ is an integer. You may not have to use all entries in the table.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| pdt | 1.0 | | | | | | |
| tmp | 3.0 | | | | | | |
| n | 10 | | | | | | |

**(ii) (4 points)** What is the time complexity of this iterative implementation in terms of $n$? What is the space complexity of this iterative implementation in terms of $n$?

**(3) Queues (20 points)** We consider different array-based implementations of a queue of integers.
**(a) (12 points)** In all parts of this question, consider the following array of 6 integers, with the contents of the array as shown, with the starting index of the array being 0.

| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 6 | 7 | 5 | 1 | 2 | 3 |

Each of following questions **(i)–(iv)** presents an array-based implementation of a queue. For each question, you have to decide whether the implementation is correct. You have to justify your answer.

In each implementation, we assume that the `front` and `rear` indices were both initialized to 0 at the beginning to indicate an empty queue. After some enqueueing and dequeueing operations, we now have the array as shown above.

**(i) (4 points)** In implementation I, `front` is now 2, `rear` is 5, and the integers in the queue (from front to rear) are 5, 1, 2, 3.

**(ii) (4 points)** In implementation II, `front` is now 2, `rear` is 5, and the integers in the queue (from front to rear) are 5, 1, 2.

**(iii) (4 points)** In implementation III, `front` is now 2, `rear` is 5, and the integers in the queue (from front to rear) are 1, 2, 3.

**(iv) (4 points)** In implementation IV, `front` is now 2, `rear` is 5, and the integers in the queue (from front to rear) are 7, 6, 3.

**(b) (4 points)** Consider the same array of 6 integers in **(a)**, with the contents of the array as shown:

| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 6 | 7 | 5 | 1 | 2 | 3 |

Assume that the starting index of the array is 0. Now, this array is being used as a priority queue, where the entries are maintained in ascending order from front to rear. We assume that the `front` and `rear` indices were both initialized to 0 at the beginning. After some enqueueing and dequeueing operations, we now have the array as shown above, where the priority queue is **full** and has the following valid entries (from front to rear): 1, 2, 3, 6, 7.

Now, you want to enqueue 4 into the priority queue. The enqueue operation involves the following steps: (1) Double the size of the array to accommodate the growth; (2) Make the fewest number of data moves to turn the expanded array into a valid queue; and (3) Insert 4 into the priority queue.

Show the contents of the priority queue after each of the three steps. For entries whose content are not known, use '?' for those entries.

After step (1):

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |

After step (2):

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |

After step (3):

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |