ECE368 Exam 1
Spring 2018

Thursday, February 22, 2018, 6:30-7:30pm
LILY 1105

Sign the following statement (AND write down your name). If the statement is not signed, the exam will not be graded and it will not be returned.

I pledge on my honor that I have neither given nor received any unauthorized assistance on this exam. If I fail to honor this pledge, I will receive a failing grade for this course and be subject to disciplinary action.

Signature:

First Name (Given Name):

Last Name (Family Name):

This is an *open-book, open-notes* exam. Electronic devices are not allowed.

This exam consists of 8 pages; it is *your responsibility* to make sure that you turn in a complete copy of the exam.

If you score 50% or more for a question, you are considered to have met all learning objectives associated with that question. Learning objectives 1–3 are covered in this exam.

Be *concise*. When you are asked of the time complexity of an algorithm, it is *not* necessary to do a line-by-line analysis of the algorithm. *A general explanation would suffice*.

Assume that all necessary "`.h`" files are included and all `malloc` function calls are successful.

If you are not clear about what a question is asking, you can explain what you are answering (e.g., "I think this question is asking for ...") or you can state assumptions that you are making (e.g., "I assume that the entry -1 is equivalent to NULL").

Your Purdue ID should be visible for us to verify your identity.

If you finish the test before 7:25pm, you may turn in the test and leave. **After 7:25pm, you have to stay until we release the entire class. Stop writing at 7:30pm. If you continue to write, that is considered cheating.**

When we collect the copies of test, you are to remain in your seat in an orderly manner until we have collected and counted all copies.

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1 Number Representation (20 points) (Learning Objective 2)

The following function prints a given unsigned long integer `n` to the `stdout` in a specified number base (radix `r`). The radix `r` may be any integer between 2 and 10 (inclusive).

For a radix `r`, the valid symbols are `0` through `r-1` (inclusive). For a symbol of value in {`0, ..., 9`}, it is printed as a character in {`'0', ..., '9'`} accordingly.

```
1  // express an unsigned number n using symbols for a radix r,
2  // where 2 <= r <= 10, e.g.,
3  // r = 2, symbols are '0', '1'
4  // r = 3, symbols are '0', '1', '2'
5  // r = 10, symbols are '0', ..., '9'
6  //
7  void print_unsigned_long(unsigned long n, unsigned int r)
8  {
9     unsigned long weight = 1;     // weight is r^0
10    while ((n / weight) >= r) {   // while n/r^k >= r
11       weight = weight * r;       // weight is r^k
12    }
13    while (weight > 0) {          // while r^k > 0
14       fputc('0' + (n / weight), stdout); // print symbol
15       n = n % weight;            // update n and weight for
16       weight = weight / r;       //    positions k-1 to 0
17    }
18 }
```

(a) (4 points) If `n` is 33, write down the respective values of `weight` in the form of r^k for different values of `r` when the `while`-loop in **lines 10-12** terminates. For example, 2^1 or 5^0 is of the expected format.

- `n` is 33, `r` is 2: `weight` is
- `n` is 33, `r` is 5: `weight` is

(b) (4 points) Write down in terms of `n` and `r` the number of times the following statements are executed:

- **Line 9**:
- **Line 10**:
- **Line 11**:

- **Line 13**:

(c) (4 points) Write down in terms of `n` and `r` the time complexity and space complexity of the implementation `print_unsigned_long`. Justify your answer.

- Time complexity:

- Space complexity:

(d) (4 points) The following function is a recursive implementation to print a given unsigned long integer `n` to the `stdout` in a specified number base (radix `r`).

```
19 void rec_print_unsigned_long(unsigned long n, unsigned int r)
20 {
21    if (n < r) {  // n can be represented with one symbol
22       fputc('0' + n, stdout); // print symbol for n
23       return;
24    }
25    rec_print_unsigned_long(n / r, r); // print symbols for n/r
26    fputc('0' + (n % r), stdout);      // print symbol for n%r
27 }
```

Draw a computation tree showing the recursive calls when `n` is 33 and `r` is 3. In each node of the computation tree, you have to show the values of `n` and `r` for that function call.

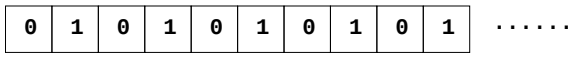
(e) (4 points) Write down in terms of `n` and `r` the time complexity and space complexity of the implementation `rec_print_unsigned_long`. Justify your answer.

- Time complexity:

- Space complexity:

2 Shell Sort (20 points) (Learning Objectives 2 and 3)

(a) (10 points) Consider the following example array, and the sequence $\{1, 2, 4, ..., 2^p\}$ for Shell sort. Assume that there are n elements in the array, and 2^p is the largest power of 2 that is less than n.



(a)(i) (2 points) For **each** of the 2^p-sorting, ..., 4-sorting, and 2-sorting, what is the time complexity in terms of n? Justify your answer.

ANS:

(a)(ii) (2 points) What is the **total** time complexity of performing 2^p-sorting, ..., 4-sorting, and 2-sorting? Justify your answer.

ANS:

(a)(iii) (4 points) What is the time complexity of performing 1-sorting? What is the **overall** time complexity of performing Shell sort using the sequence $\{1, 2, 4, ..., 2^p\}$? Justify your answer.

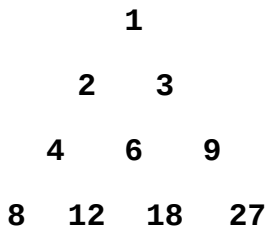
ANS:

(a)(iv) (2 points) Provide an example where the time complexity of Shell sort is as high as that in **(a)(iii)** when the sequence used is $\{1, 3, 9, ..., 3^q\}$, where 3^q is the largest power of 3 that is less than n.

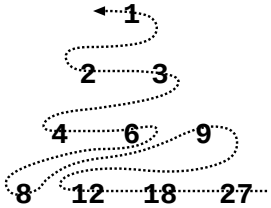
ANS:

(b) (10 points) Consider a Shell sort routine that uses the sequence of numbers of the form $2^p 3^q$, where $p, q \geq 0$ for sorting.

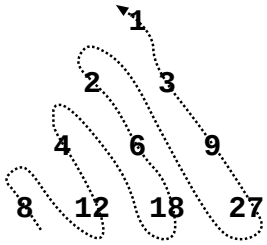
(b)(i) (4 points) Suppose the routine is being applied to an array of 110 elements. Complete the following figure by generating all appropriate numbers < 110 that belong to the sequence. The numbers that you fill in may not form complete rows.



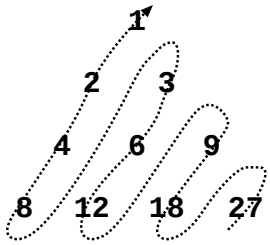
(b)(ii) (4 points) The sequence that allows $O(n(\log n)^2)$ time complexity is one that has the numbers of the form $2^p 3^q$ in ascending order (and Shell sort performs k-sortings using these numbers in descending order). The following figures shows such an ordering of k-sortings, where we perform 27-sorting, 18-sorting, 12-sorting, 9-sorting, 8-sorting, and so on, until 1-sorting. Note that in general, the sequence could have more numbers and the rows may not be complete, as in your answer for **(b)(i)**. For illustration purpose, we show the ordering using only four complete rows.



The following shows two other orderings (I and II) in which we can perform Shell sort. In ordering (I), for example, we perform 8-sorting, followed by 12-sorting, 4-sorting, and so on, until 1-sorting.



(I) Left-to-right, diagonally bottom-up



(II) Right-to-left, diagonally bottom-up

In ordering (I), the sequence $\{1,3,9,...\}$ is used eventually, and in ordering (II), the sequence $\{1,2,4,...\}$ is used eventually. Does performing Shell sort using any of these two orderings have $O(n(\log n)^2)$ time complexity or the same time complexity as in **(a)(iii) and (a)(iv)**? Justify your answer.

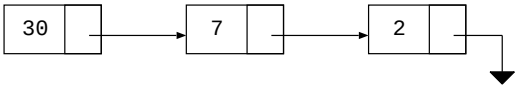
ANS:

(b)(iii) (2 points) Between ordering (I) and ordering (II) in **(b)(ii)**, which is more efficient? Justify your answer.

ANS:

3 Linked Lists (20 points) (Learning Objective 1)

Consider the following linked list. For this question, you should assume that the **operations associated with a stack, queue, or priority queue are all implemented with best efficiency (in terms of run time and memory requirement), taking into account the properties of a linked list, as covered in the class.**



(a) (4 points) The linked list at the beginning of this question is being used as a **stack**. We apply the following two operations on the stack **consecutively**. Show the resulting linked list after each operation.

Push(Stack, 5):

Pop(Stack):

(b) (4 points) The linked list at the beginning of this question is being used as a **queue**. We apply the following two operations on the queue **consecutively**. Show the resulting linked list after each operation.

Enqueue(Queue, 5):

Dequeue(Queue):

(c) (4 points) The linked list at the beginning of this question is being used as a **priority queue** (PQ). **You have to deduce the definition of priority based on the given list.** We apply the following two operations on the PQ **consecutively**. Show the resulting linked list after each operation.

PQ_Enqueue(PQ, 5):

PQ_Dequeue(PQ):

(d) (8 points) Consider an array-based implementation of singly linked list(s). The contents of the array implementing the linked list(s) are as follows:

index	info	next
9	109	5
8	30	0
7	107	9
6	2	-1
5	105	4
4	104	-1
3	103	7
2	102	3
1	101	2
0	7	6

`Unused_List index: 1`

Among the linked lists in the array, one maintains a list of unused nodes. We refer to that list as `Unused_List`, whose index at the moment is at position 1. The other list is the one we showed at the beginning of the question.

Whenever a new node is required by some other functions (push, enqueue, or PQ_enqueue), the node is obtained from `Unused_List` (and the index is updated). Whenever a node is deleted from a linked list and is no longer required (pop, dequeue, or PQ_dequeue), it is returned to `Unused_List` (and the index is updated). Again, **you should assume that the** `Unused_List` **is maintained with best efficiency.**

(d)(i) (2 points) Draw the `Unused_List`.

ANS:

(d)(ii) (6 points) Update in the next page the contents of the array and the `Unused_List` index **after the consecutive executions** of **PQ_Enqueue(PQ, 5)** and **PQ_Dequeue(PQ)** in **(c)**. **All you have to do is to cross out entries that should be updated and replace them with correct values.**

PQ_Enqueue(PQ,5)

index	info	next	
9	109	5	
8	30	0	
7	107	9	
6	2	-1	`Unused_List index: 1`
5	105	4	
4	104	-1	
3	103	7	
2	102	3	
1	101	2	
0	7	6	

PQ_Dequeue(PQ)

index	info	next	
9	109	5	
8	30	0	
7	107	9	
6	2	-1	`Unused_List index: 1`
5	105	4	
4	104	-1	
3	103	7	
2	102	3	
1	101	2	
0	7	6	