ECE368 Exam 1
Spring 2017

Wednesday, February 22, 2017
6:30-7:30pm
CL50 224

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, and that I will not provide help to any other student. 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:`

`Signature:`

If the statement is not signed, the exam will not be graded and it will not be returned.

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

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.

This exam consists of 7 pages; it is *your responsibility* to make sure that you turn in a complete copy of the exam.

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").

If you score 50% or more for a question, you are considered to have met all learning objectives associated with that question.

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. Shell Sort (20 points) (Learning Objectives 2 and 3) The function `shellsort` sorts integers stored in `array`, whose `size` is also given. The `shellsort` function calls the function `generate_sequence` to generate a sequence used for Shell sorting. Given the `size` of the `array` to be sorted, the `generate_sequence` function stores the sequence in a dynamically allocated array called `sequence`. The parameter `*seq_size` stores the location of the last integer in `sequence`. If `*seq_size` is -1, the array `sequence` is empty.

```
1  int *generate_sequence(int size, int *seq_size)
2  {
3     int powers_3 = 1; // powers of 3
4     int p = 0;        // exponent of powers of 3
5     while (powers_3 < size) { // terminates when powers of 3 >= size
6        powers_3 *= 3;
7        p++;
8     }
9
10    int *sequence = (int *)malloc(sizeof(int)*p*(p+1));
11
12    int seq_count = -1; // location of last int in sequence
13    int next_idx = 0;   // location to generate next int in sequence
14    int k = 1;          // next int in sequence
15    powers_3 = 3;       // next powers of 3 in sequence
16    while (k < size) {  // terminates when k >= size
17       seq_count++;     // increment seq_count
18       sequence[seq_count] = k; // save k in sequence
19       if (sequence[next_idx]*2 > powers_3) {  // determine the next k
20          k = powers_3;             // k is some 3^q
21          powers_3 *= 3;
22       } else {                     // or
23          k = sequence[next_idx]*2; // k is 2 * some int in sequence
24          next_idx++;               // i.e., k is some 2^p 3^q
25       }
26    }
27    *seq_size = seq_count;
28    return sequence;
29 }
30
31 void shellsort(int *array, int size)
32 {
33    int seq_size;
34    int *sequence = generate_sequence(size, &seq_size);
35    for (; seq_size >= 0; seq_size--) { // step through int in sequence
36       int gap = sequence[seq_size];    // in reverse order
```

```
37          int j;
38          for (j = gap; j < size; j++) {
39              int temp = array[j];
40              int i = j - gap;
41              while (i >= 0 && temp < array[i]) {
42                  array[i + gap] = array[i];
43                  i = i - gap;
44              }
45              array[i + gap] = temp;
46          }
47      }
48      free(sequence);
49  }
```

(a) (2 points) What are the values of p after running lines 3–8 when the following sizes are passed into the function `generate_sequence`?

- `size` = 1:
- `size` = 4:
- `size` = 10:
- `size` = 30:

(b) (4 points) Let n be the `size` passed into the function `generate_sequence`. What is the time complexity of running lines 3–8 in terms of n using the big-O notation? Justify your answer.

(c) (6 points) Write down the integers stored in the array `sequence`, from the lowest indexed position to the highest indexed position, when the following sizes are passed into the function `generate_sequence`? If the sequence is empty, write down "empty" as your answer.

- `size` = 1:
- `size` = 4:
- `size` = 10:
- `size` = 30:

(d) (4 points) Let n be the `size` passed into the function `generate_sequence`. What is the time complexity of running lines 12–26 in terms of n using the big-O notation? Justify your answer.

(e) (4 points) Consider an array of 22 integers:

```
int array[] = {21, 20, 19, 18, 17, 16, 15, 14, 13, 12,
               11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

The array is passed to the function `shellsort` as follows:

```
shellsort(array, 0, 21);
```

Assume that the first gap that is used for Shell sorting is 18. Show the contents of `array` right after 18-sorting.

2. Lists and Trees (20 points) (Learning Objectives 1 and 2) The following code fragment defines a structure for storing a binary tree and a function for re-constructing a binary search tree from an array of integers. The integers in the array are ordered according to the postorder traversal of the original binary search tree. We also assume that all integers are distinct.

```
1  typedef struct _tnode {
2     int value;
3     struct _tnode *left, *right;
4  } tnode;
5
6  tnode *build_bst_from_postorder(int *array, int lb, int ub)
7  {
8     if (lb > ub) {
9        return NULL;
10    }
11    tnode *node = malloc(sizeof(tnode));
12    node->value = array[ub];
13
14    // find the left subtree and the right subtree of node in the array
15    // assume that all integers in left subtree <= node->value
16    int partition_idx = lb;
17    while ((partition_idx < ub) && (array[partition_idx] <= array[ub])) {
18       partition_idx++;
19    }
20    node->left = build_bst_from_postorder(array, lb, partition_idx-1);
21    node->right = build_bst_from_postorder(array, partition_idx, ub-1);
22    return node;
23 }
```

(a) (4 points) Consider an array of 12 integers:

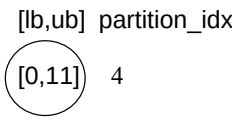
```
int array[] = {2, 4, 6, 8, 12, 16, 14, 20, 24, 22, 18, 10};
```

The integers are ordered such that they correspond to the postorder traversal of a binary search tree. The array is passed to the function `build_bst_from_postorder` as follows:

```
tnode *root = build_bst_from_postorder(array, 0, 11);
```

Draw the binary search tree re-constructed by the function (in the next page).

(b) (4 points) Complete the computation tree that corresponds to the recursive calls of the function `build_bst_from_postorder` in **2(a)**. The node that corresponds to the first call is shown below. The interval in the node corresponds to the parameters `lb` and `ub` passed to the function. The integer outside the node corresponds to the value saved in the variable `partition_idx` after executing lines 16–19 of the function `build_bst_from_postorder`. You computation tree must show the parameters `lb` and `ub` in each node. It is optional to show the variable `partition_idx`.



(c) (6 points) Let n be the number of integers in the array passed to the function `build_bst_from_postorder`. (i) What is the worst-case space complexity for the re-construction of the entire binary search tree in terms of n using the big-O notation? (ii) What is the best-case space complexity? Justify your answers. Your answers should not include the space required for the input array and the re-constructed binary search tree.

(i) Worst-case space complexity:

(ii) Best-case space complexity:

(d) (6 points) Let n be the number of integers in the array passed to the function `build_bst_from_postorder`. (i) What is the worst-case time complexity for the re-construction of the entire binary search tree in terms of n using the big-O notation? (ii) What is the best-case time complexity? It is important to take into account the time complexity in executing lines 16–19 of the function. Justify your answers.

(i) Worst-case time complexity:

(ii) Best-case time complexity: