ECE-250 – Algorithms and Data Structures (Winter 2012)

Practice Problems (Midterm)

Disclaimer: Please do keep in mind that this problem set does not reflect the exact topics or the fractions of each of the topics or the amount of work that will appear in the midterm. It is supposed to help you prepare for the midterm, but I'm including here more problems in the topics that were not covered in the assignments, and also a little bit more emphasis on the areas that seemed problematic, or where several students have reported difficulty working through it.

Disclaimer 2: I may or may not post solutions.

Mathematical background

1 – Prove that if a number n has the property that $n \bmod 4 = 3$, then the last two bits (the least-significant) are 11.

2 – Consider an algorithm `enclose_polygons` that receives a set of n simple polygons $\{P_1,\ P_2\ \cdots\ ,P_n\}$ as input (i.e., polygons that have no edge intersections and that form a closed path defining a region — see first paragraph and examples in `http://en.wikipedia.org/wiki/Simple_polygon` if you need further clarification).

The algorithm outputs a sequence of polygons $P_{k_1},\ P_{k_2},\ \cdots\ ,P_{k_n}$ where the polygon P_{k_i} is entirely contained inside polygon $P_{k_{i+1}}$ $\forall\ 1 \leqslant i < n$. That is, it outputs the same set of input polygons, but in sequence such that every polygon is entirely contained in the next polygon in the sequence.

If creating such a sequence is not possible (e.g., if there are polygons that are not entirely contained in either one of the remaining polygons), then the algorithm outputs null.

Prove that the worst-case runtime of algorithm `enclose_polygons` is $\mathbf{\Omega}(n \log n)$. That is, prove that this algorithm can not have a worst-case runtime better than $\Theta(n \log n)$.

3 – Prove by induction that $\displaystyle\sum_{k=1}^{n} k\, 2^k \ =\ 2(n\, 2^n - 2^n + 1)$

Asymptotic and Algorithm Analysis

4 – Logarithms and exponentials:

(a) Explain why all logarithms of different bases are Θ of each other.

(b) Show that the above property does not apply to exponentials (that is, explain why exponentials with different base are not Θ of each other).

(c) Prove, using the formal definition of Θ) that all logarithms of a polynomial are $\Theta(\log n)$. That is,

$$\log\left(a_k n^k + a_{k-1} n^{k-1} + \ \cdots \ + a_1 n + a_0\right) \ = \ \Theta(\log n)$$

for any degree k—you may assume that all $a_i > 0$, although it is enough to assume that $a_k > 0$.

(d) Explain why the above property does not apply to exponentials.

5 – What are the run times of the following loops? (in all cases, assume that n is sufficiently large if it needs to be assumed for the problem to make sense)

```
for (int i = 0; i < n*n; i++)
{
    sum += i;
}

for (int i = 0; i*i < n; i++)
{
    sum += i;
}

for (int i = n*n; i > 1; i /= 2)
{
    if (i > 16)
    {
        sum += i;
    }
}

for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        for (int k = j+1; k < n; k++)
        {
            sum += i*j*k;
        }
    }
}
```

6 – Assuming that the function `binary_search` does indeed a binary search for `value` in the given array:

```
int binary_search (int value, int * array, int array_size)
```

Notice that the function `binary_search` looks for the position where a value is or would be in the given array. The emphasis being on the fact that if the value to be searched is not found in the array, then the function returns the position where it would go if it were to be inserted.

Examples: if an array with size 8 contains the values 3, 5, 8, 11, 18, 29, 30, 45 (3 being at position 0, 45 being at position 7, as per normal C++ convention on arrays subscripts), then a binary search for the value 8 would return 2, a binary search for the value 13 would return 4, and a binary search for -3 would return 0.

Describe what the following function does, and determine its run time:

```
    // assume both arrays have size n
void mystery_function (int * output_array, const int * array, int n)
{
    for (int i = 0; i < n; i++)
    {
        const int pos = binary_search (array[i], output_array, i);
            // search the value given in the first parameter in the
            // array given by second parameter, with size = third parameter
        for (int k = i; k >= pos; k--)
        {
            output_array[k+1] = output_array[k];
        }
        output_array[pos] = array[i];
    }
}
```

7 – Determine the run time of the following recursive function (i.e., determine the recurrence relation representing the run time, and then solve the recurrence relation), and explain the "intuition" of why we obtain such run time:

```
int mystery_function (int * array, int n)
{
    if (n == 1)
    {
        return array[0];
    }
    else if (array[0] < array[n/2] && array[n/2] < array[n-1])
    {
        return mystery_function (array, n/2);
    }
    else
    {
        return mystery_function (array + n/2, n/2);
    }
}
```

Hash Tables

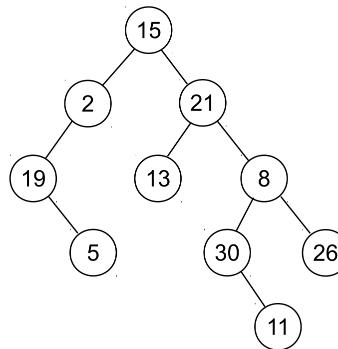
8 – Name and explain two problems that are addressed (solved) in double hashing when choosing a secondary hash function (the "jump size" hash function) that can only take odd values.

9 – Consider a hash table of size 16, to store integer values. Using a hash function obtained by adding the digits of the given value, and taking that result modulo 16, insert, using linear probing, the values 867, 167, 993, 230, 1497 and then remove 867.

Trees

10 – Write recursive C++ functions that output (as in, output to the console, with `cout`) the elements of a binary tree using pre-order, in-order, and post-order traversal. Assume a `Binary_node` definition such as the one shown in class (you can look it up in the course slides.

11 – Given the following binary tree, show the output of breadth-first, pre-order, in-order, and post-order depth-first traversals:



12 – Write a C++ function to update the stored heights in an AVL tree. The function receives the node (a pointer to `Binary_node<Type>`) where the change happened (the inserted node, or the parent of the removed node).

You may assume a definition similar to that of class template `Binary_node` shown in class, with an additional data member to hold the height of the tree rooted at the node. You may also assume that the function is a friend function of `Binary_node`. Needless to say that the function must run in $\Theta(\log n)$, where n is the number of nodes in the AVL tree.

13 – Given the tree from question 11, insert the values into an initially empty AVL tree, inserting in the order given by:

(a) Breadth-first traversal
(b) In-order depth-first traversal

14 – Given the resulting AVL tree from question 13-b, remove the root element four times.

15 – Given a weight-balanced binary search tree, we want to store in each node the weight of the tree rooted at the node (similar to what we do with AVL trees with the height). Write a C++ function to update these stored weights when the tree is changed. The function receives a pointer to the inserted node, or the parent of the removed node (same comments as in second paragraph of question 12).