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

Suggested solutions for some of the problems

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.

Solution:

This is similar to the argument in class (talking about double hashes) about how a number is odd if and only if its LSB is 1. Here, we also go for a direct proof, proceeding as follows: consider a number n and its m-bits binary representation:

$$n = \sum_{k=0}^{m} b_k 2^k$$

We're interested in proving something about the last two digits (the LSB), so it makes sense that we pull those out from the summation:

$$n = \sum_{k=0}^{m} b_k 2^k \;=\; b_0 + 2b_1 + \sum_{k=2}^{m} b_k 2^k \;=\; b_0 + 2b_1 + 2^2 \sum_{k=2}^{m} b_k 2^{k-2}$$

Since b_0 and b_1 can only take values 0 or 1, then clearly, $b_0 + 2b_1 < 4$, so we can write $n = 4 \cdot S + r$, where $r \triangleq b_0 + 2b_1$ and S is the sum in the above equation.

Since $0 \leqslant r < 4$, then clearly the remainder of the division of n by 4 is precisely r (if you don't like the "informality" of this argument, you probably remember the *Division Algorithm* from your Discrete Math course?). Since $n \bmod 4 = 3$, we directly obtain the result ($r = 3$, and r has binary representation given by the two least-significant bits of n, so these two bits must be both 1)

2 – Consider an algorithm `enclose_polygons` that receives a set of n simple polygons $\{P_1,\ P_2\ \cdots\ , P_n\}$ as input (see first paragraph and examples in `http://en.wikipedia.org/wiki/Simple_polygon`).

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 $\Theta(n \log n)$.

Solution:

Not surprisingly, we prove the statement by reduction; specifically, reduction from sort. We know that sorting n arbitrary values takes $\mathbf{\Omega}(n \log n)$, so we will reduce sorting to `enclose_polygons` (with

a linear time reduction), showing that if `enclose_polygons` could execute faster than $n \log n$, then we would have a sorting algorithm faster than $n \log n$, which we know is not possible.

The reduction is quite straightforward: given n values $\{x_1, x_2, \cdots, x_n\}$, we construct n polygons that are all squares. That is, polygon i is given by the vertices $P_i = \{(x_i, x_i), (-x_i, x_i)\,(-x_i, -x_i)\,(x_i, -x_i)\}$

We feed these polygons to algorithm `enclose_polygons`, relying on the fact that larger values of x_i produce polygons that enclose those produced by the smaller values of x_i; thus, the output gives us the polygons in ascending order by their sides, meaning that we can directly obtain the input values in ascending order.

The reduction clearly takes linear time, so it does prove that `enclose_polygons` has run time $\mathbf{\Omega}(n \log n)$.

This last detail is always important in a proof by reduction: if we denote the time for our sorting-in-terms-of-enclose_polygons algorithm by $\mathrm{T_S}(n)$, the run time of `enclose_polygons` by $\mathrm{T_{EP}}(n)$ and the run time of the reduction by $\mathrm{T_R}(n)$, then, since $\mathrm{T_S}(n) = \mathrm{T_R}(n) + \mathrm{T_{EP}}(n)$ (since our sorting algorithm involves the reduction and an invocation to `enclose_polygons`), then we know that

$$\mathrm{T_R}(n) + \mathrm{T_{EP}}(n) \geqslant n \log n \tag{1}$$

but then, if $\mathrm{T_R}(n) \geqslant n \log n$, then Eq.(1) holds regardless of the run time of `enclose_polygons`, and so that statement would say absolutely nothing about such run time.

If, however, $\mathrm{T_R}(n) < n \log n$, then Eq.(1) does imply that $\mathrm{T_{EP}}(n) \geqslant n \log n$ (what we want to prove). (we should use the appropriate Landau symbols in Eq.(1) and the following lines, instead of inequality symbols — I wrote it like this to make the idea more clear).

Note: there is a (not-so-subtle) detail that ruins the above argument — I'll leave it up to you to figure out that detail, and to fix the proof (I estimate that that the more challenging part is realizing what the problem is; once noticing, what to do to fix the proof follows quite immediately).

Asymptotic and Algorithm Analysis

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;
            }
        }
    }
```

Solution:

In all cases, we observe that everything inside the loop is $\Theta(1)$ (since they're just arithmetic operations, which we know execute in constant time). Also, the condition of the loops are all constant-time expressions.

The first one is clearly $\Theta(n^2)$ (the loop executes n^2 times); the second loop executes $\sqrt{n}$ times, so we get $\Theta(\sqrt{n})$.

For the third one, since at each pass of the loop we divide i by 2, then after k passes, $i = \frac{n^2}{2^k}$. The loop stops when $i = 1$, or $n^2 = 2^k \;\Rightarrow\; k = \lg n^2$. That is, the loop executes $\lg n^2$ times ($= 2 \lg n$), thus, we get $\Theta(\log n)$

For the fourth one, going from the innermost loop and working our way out, we observe that the innermost loop executes $n-j-1$ times; that is, the first pass of the middle loop, we execute $n-1$ times, then $n-2$ times, then $n-3$ and so on. If we add these together, we obtain $1+2+3+ \cdots +(n-2)+(n-1)$, which we recognize (right?) as $\frac{n(n-1)}{2}$. So, the middle loop executes in $\Theta(n^2)$; since that one is executed n times, then we get a total run time of $\Theta(n^3)$

Trees

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).

Solution:

Two key details here (I'll just mention the two details — you should be able to work on the solution):

(1) The only nodes for which the height can change are the ones on the path from the root node to

the node where the change happened.

(2) The height for those nodes *can* change (it does not necessarily change).

The reason for (1) is quite simple: For each sub-tree, if a node is inserted in one of its sub-trees, then the height for the other sub-tree can not change. Since this applies to all sub-trees in the tree, it is clear that only the nodes in the path from the root node to the point of change are the ones that can be affected.

The reason for (2) is that the height for a node is one plus the larger of the heights of the two sub-trees. If a node was added in the sub-tree that had the lower height, then there will be no change (since the height is determined by the other sub-tree).

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).

Solution:

Again, I'll just mention the key detail here:

Unlike for question 12, here we are storing weights, and the weight of a tree is 1 plus the sum of the weights of the two sub-trees; that means that the weight for all the nodes in the path from the root node to the added or removed node *do change*. If we added a node, all of the weights in the path from the root to it are increased by 1; and if removed, the all the weights in the path from root to the parent of the deleted node are decreased by 1.