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

Tutorial 2 (2012-01-26) – Suggested solutions

1 - Assuming well-behaved positive functions such as those that we encounter when analyzing algorithms, show that:

(a) Little-oh defines a partial ordering.
(b) Big-Oh defines a weak ordering.
(c) Big-Theta is an equivalence relation (one of the three conditions was already done during last week's tutorial session).

Solution:

(a) The relationship little-oh is antisymmetric; if $\mathrm{f}(n) = o(\mathrm{g}(n))$, then $\mathrm{g}(n) \neq o(\mathrm{f}(n))$.

It is transitive: if $\mathrm{f}(n) = o(\mathrm{g}(n))$ and $\mathrm{g}(n) = o(\mathrm{h}(n))$, then there exist $N_1 > 0$ and $N_2 > 0$, and N being the larger of N_1 and N_2 such that for every $c > 0$, we have:

$$0 \leqslant \mathrm{f}(n) \leqslant c\mathrm{g}(n) \quad \forall\ n > N$$

and

$$0 \leqslant \mathrm{g}(n) \leqslant c\mathrm{h}(n) \quad \forall\ n > N$$

Which means that

$$0 \leqslant \mathrm{f}(n) \leqslant c^2\mathrm{h}(n) \quad \forall\ n > N$$

This shows that $\mathrm{f}(n) = o(\mathrm{h}(n))$ (for every $c' > 0$, this holds, since the two other conditions must hold for $c = \sqrt{c'}$ — since they must hold for every $c > 0$)

Lastly, the relation does not exhibit totality — it is not the case that for every $\mathrm{f}(n)$ and $\mathrm{g}(n)$, it must be the case that either $\mathrm{f}(n) = o(\mathrm{g}(n))$ or $\mathrm{g}(n) = o(\mathrm{f}(n))$ ($\mathrm{f}(n) = \mathrm{g}(n)$ is a trivial example to show that this is the case; more in general, any functions that are Θ of each other are a valid example)

(b) For this, we need the condition that the functions are nice and well-behaved; otherwise, we could find pairs of functions such that neither one is big-Oh of the other one (meaning that no total order is possible).

Assuming that, this is clearly a linear or total order for the equivalence classes given by the functions that are Θ of each other.

(c) The relationship is obviously reflexive ($\mathrm{f}(n) = \Theta(\mathrm{f}(n))$; trivially true with $c_1 = c_2 = 1$ in the definition).

We already showed that it is symmetric (last week's tutorial)

It is transitive:

$$\mathrm{f}(n) = \Theta(\mathrm{g}(n)) \ \Rightarrow\ \exists\ c_1 > 0, c_2 > 0, N_1 > 0 \ \mid\ 0 \leqslant c_1\mathrm{g}(n) \leqslant \mathrm{f}(n) \leqslant c_2\mathrm{g}(n) \ \ \forall\ n \geqslant N_1$$

$$\mathrm{g}(n) = \Theta(\mathrm{h}(n)) \ \Rightarrow\ \exists\ c_1' > 0, c_2' > 0, N_2 > 0 \ \mid\ 0 \leqslant c_1'\mathrm{h}(n) \leqslant \mathrm{g}(n) \leqslant c_2'\mathrm{h}(n) \ \ \forall\ n \geqslant N_2$$

From these two conditions, if we let N be the larger of N_1 and N_2, we see that

$$0 \leqslant c_1 c_1' \mathrm{h}(n) \leqslant \mathrm{f}(n) \leqslant c_2 c_2' \mathrm{h}(n) \quad \forall\, n \geqslant N$$

Showing that $\mathrm{f}(n) = \Theta(\mathrm{h}(n))$

2 - Determine the run time (worst-case only) of the following operations:

(a) Assuming that there are *always* at least one negative value and at least one positive value in the array, find the first negative value in an array of:
(i) Random values
(ii) Ordered values in ascending sequence
(iii) Ordered values in descending sequence

(b) Same as part (a), but for a linked list

(c) Insert an element into a singly-linked list with random values:
(i) After a given element (i.e., you're given a pointer to a node)
(ii) Before a given element
(iii) Exactly at the middle (plus/minus rounding if the list contains an odd number of elements)

Solution:

(a-i) Worst-case is clearly $\Theta(n)$, as we may need to traverse the entire array to find the negative value in the last position.

(a-ii) Worst-case is clearly $\Theta(1)$; since we have the guarantee that there is at least one negative value, when ordered in ascending sequence the first element will necessarily be negative.

(a-iii) Since the values are in order, we can always use binary search (even if not looking for a specific value, we can still use binary search), so we get $\Theta(\log n)$.

Do notice that the argument that we start from the end (which we can, since it is an array) and thus we find it in constant time is not at all valid; if we search backwards, we find the *last* negative number; we don't know how many there are, and so we don't know how many times we need to move backwards.

(b) The only difference is in (iii), since the linked list does not accept binary search. Thus, regardless of searching foraward or backwards (in case it is a doubly-linked list), worst-case will necessarily be $\Theta(n)$; if we start at the beginning and search forward, the only negative value might be at the end; and if we start at the tail and search backwards, the worst-case is a single non-negative value at the first position.

(c-i) This is clearly constant time (we do that in a fixed number of constant-time operations)

(c-ii) The only way to accomplish this is to start at the beginning and move forward until we reach the element before the one we're given; worst-case occurs when the given element is the last, thus we have $\Theta(n)$

(c-iii) Since we do not have random access, we need to start at the beginning and "count" our way to the middle element (advance $n/2$ times). Depending on whether or not we can obtain the size of the list in constant time, we may need a complete scan of the list to determine the size first. Either way,

we have $\Theta(n)$ always (it does not depend on the data).

3 - Compare (a discussion suffices; no need to formally determine the actual figures) the run times (if applicable, discuss the "actual time" performance) of operations on a queue (enqueue and dequeue) when we implement it in terms of:

(a) An array (linear array)
(b) A circular array or circular buffer
(c) A linked list

Solution:

(a) A linear array requires either continuously reallocating, if we keep track of beginning and end of the queue, or continuously shifting all the elements (each time we dequeue). And even then, if we do many enqueues and very few dequeues, we may need to reallocate to accommodate for the queue's growth. This is a non-trivial probabilities problem, so we'll be happy with an intuition-based rough estimate of $\Theta(n)$ either way.

The enqueue and dequeue operations themselves are, however, most efficient when using a linear array — directly access an element by subscript, and update one variable.

(b) With a circular array, no shifting of elements is required; reallocation might still be needed if enqueues and dequeues are not balanced. Though access to elements and updates of the begin and end indicators are still constant time, they are slower, in "actual time" performance, than for a linear array, since we require to do arithmetic, including a modulo operation.

Reallocations are no different than with respect to the linear array.

(c) The big advantage with a linked list is that we don't worry about reallocations — the queue grows naturally as the linked list does, with constant time for each operation. We do not need to keep pointers or indexes for begin or end, since these are implicitly present for a linked list.

We do, however, have storage overhead — there is a per-element $\Theta(1)$ storage — compared to either version using arrays.

4 - (If enough time) Determine the run time of the following function, operating on a polygon (a sequence of points) represented as an array of point objects (we have some class Point to represent the points/vertices of the polygon):

The function needs to check a condition for every pair of points; the condition is symmetric (i.e., if the condition is met for points p_i,p_j, then it is also met for the pair p_j,p_i), and it is checked in constant time. It is also known that the condition is never met between a point and the five points that follow.

Determine the runtime of a function meeting the above requirements.

Solution:

There are two approaches here. The basic principle being that you loop through the points in the array, and for each point p_i, you check the condition with the points that follow only (since the condition is symmetric, so the pairs p_k, p_i for $k < i$ were already checked in previous iterations of the

loop).

So, if we use nested loops, the outer loop going from point 1 to point $n-1$ and the inner loop going from point $i+1$ to point n (where i is the current iteration for the outer loop), then we get an arithmetic sum (first iteration, n checks, second iteration, $n-1$ checks, then $n-2$, and so on). This is $\Theta(n^2)$

But there is one additional optimization: we do not need to check the condition for the 5 points that follow (i.e., at iteration i, we should check points $i+6$ to n). That means five fewer iterations of each execution of the inner loop with respect to the unoptimized form, and of course, the outer loop runs five times fewer, since we only need to get to $n-6$. So, this is $5(n-6)$ fewer times than the unoptimized form, which is $\Theta(n^2)$, and the result is clearly $\Theta(n^2)$.

The other approach would be simply to do the sums exactly; we would get something like an arithmetic sum going from 1 to $n-5$, and that would be $(n-5)(n-4)/2$, and we see that this is $\Theta(n^2)$.