COMP4500/7500 Advanced Algorithms & Data Structures
Sample Solution to Tutorial Exercise 8 (2014/2)[*]

School of Information Technology and Electrical Engineering, University of Queensland

September 13, 2014

1. (Kingston exercise 3.6)

   Consider the following algorithm for adding one to a binary number, represented as an array, A, of k bits, assuming that there is no overflow. The indices of A range from 0 to $k-1$. The number I represented by the array is given by the formula

$$I = \sum_{i=0}^{k-1} A[i]2^i.$$

```
INCREMENT($A, k$)
1  $i = 0$
2  while $i < k$ and $A[i] == 1$
3      $A[i] = 0$
4      $i = i + 1$
5  if $i < k$
6      $A[i] = 1$
```

The algorithm is clearly $O(k)$ in the worst case. Given that the array A is initially set to all 0's, show that a sequence of n increment operations is $O(n)$ by showing that INCREMENT has an amortised complexity of $O(1)$. One method of doing this is already provided in the lecture notes. You should do it using the Potential Method. Thus, choose a suitable potential function that allows the actual cost of the **while** loop to be cancelled out by the change in potential.

Sample solution. We count the number of times the test in the **while** loop is evaluated. The comparison is executed once for each consecutive 1 at the low end of the array, and a final time for the first 0 entry in the array. Because we cannot predict the number of times the loop will be executed, we need to arrange for the actual cost of each loop execution (i.e., 1) to be cancelled out by a decrease in the potential function of 1 as well. Each time the loop comparison is successful the corresponding bit is changed from 1 to 0 in the body of the loop. This suggests choosing the number of bits that have the value 1 as the potential function for our amortised analysis:

$$\Phi(A) = \sum_{i=0}^{k-1} A[i].$$

This potential function has the desired property that, for all arrays A,

$$\Phi(A) \geq \Phi(A_0) = 0,$$

where A_0 is the initial all 0 value of A.

The amortised cost of an INCREMENT operation can now be calculated. If there are m low order 1 bits in the array A then the loop comparison is made $m+1$ times, giving an actual cost of $m+1$. During the course of the INCREMENT operation the m low order bits of A are changed from 1 to 0 (a reduction in potential of m), and the next bit is changed from 0 to 1 (an increase of 1). Hence the change in potential is $1-m$. The amortised cost of the operation is the sum of the actual cost and the change in potential:

$$a = (m+1) + (1-m) = 2.$$

As the amortised cost of a single INCREMENT operation is $O(1)$, the amortised cost of a sequence of n operations is $O(n)$.

[*]Copyright © reserved September 13, 2014

2. (Kingston exercise 3.7)

   Given a set of n points in the plane, a **convex hull** is a sequence of points from the set which defines a convex polygon enclosing all the points in the set. One way of finding the convex hull of a set of points begins by sorting the points by their x-coordinates. This step can be implemented to have $O(n \log n)$ complexity in the worst case. Next, the points are added one by one from left to right, to a growing convex hull.

   We assume that we are given the points, ordered on their x-coordinates. The points are presented to us as an array p of type $Entry$. The representation of each point in the convex hull initially contains just the coordinates of the point. On completion all points in the convex hull have links to the two adjacent points in the hull in the clockwise and anti-clockwise directions. (To avoid peculiar cases we specify that no two points have identical x or y coordinates. Consider what happens if we have points on the same line.)

   Our data structures use a type $Point$ which is a record with two real components: the x and y coordinates of the point; and a type $Entry$ which is a record with three components: a point pt; and two links $clock$ and $anti$ to the two adjacent points in the hull in the clockwise and anti-clockwise directions, respectively.

   Our algorithm builds the entries which represent the convex hull in the array p.

   The algorithm sets up an initial convex hull consisting of the first three points. Then the remaining points are processed from left to right (i.e., ordered on their x coordinate). At each stage a convex hull is formed from the set of points considered so far. The first time a point is considered it will be the rightmost point of the set of points considered so far. Hence it must be added to the convex hull. In the process of adding the new point it may be necessary to delete some of the points that were previously in the hull.

   The processing of each point is done in two parts: first, the point it connects to in the clockwise direction is determined, and then the point it connects to in the anti-clockwise direction is determined.

   A number of auxiliary procedures are used. The procedure LINKCLOCKWISE links two points i and j together in a clockwise direction.

```
LINKCLOCKWISE($i, j$)
1  $p[i].clock = j$
2  $p[j].anti = i$
```

   The function CONVEX determines whether the line connecting points $p1$–$p2$ and $p2$–$p3$ are convex in a clockwise direction.

```
CONVEX($p1, p2, p3$)
1  $\alpha = Angle(p1, p2)$
2  $\beta = Angle(p2, p3)$
3  **return** $(\pi - \alpha + \beta) \bmod (2 * \pi) \leq \pi$
```

   The function ANGLE(pi, pj) determines the angle between the x-axis and the line joining points pi and pj. You may assume that the worst-case time complexity of ANGLE is $O(1)$.

```
ConvexHull$(n, p)$
// Pre: $n \geq 3$ and $p$ is ordered on $x$-coordinate.
// Pre: no two points have identical $x$ or $y$ coordinates
// Set up the initial 3 point hull.
if Convex$(p[1].pt,\ p[2].pt,\ p[3].pt)$
    LinkClockwise$(1, 2)$; LinkClockwise$(2, 3)$; LinkClockwise$(3, 1)$
else // Convex$(p[1].pt,\ p[3].pt,\ p[2].pt)$
    LinkClockwise$(1, 3)$; LinkClockwise$(3, 2)$; LinkClockwise$(2, 1)$
// The remaining points are processed one at a time in increasing order of $x$-coordinate (as $p$ is
// assumed to be sorted on $x$-coordinates). A convex hull of the points processed so far is maintained.
// Each new point that is processed has an $x$-coordinate greater than all the points processed so far.
// Hence the point needs to be added to the hull. But adding the point to the hull may lead to points
// already in the hull being deleted.
for $i = 4$ to $n$
    // Determine the clockwise connection: starting from the previous rightmost point in the hull and
    // working clockwise, we determine the first point that forms a convex angle in a clockwise direction.
    $c = i - 1$
    while not Convex$(p[i].pt,\ p[c].pt,\ p[p[c].clock].pt)$
        $c = p[c].clock$
    // Determine the anti-clockwise connection: starting from the previous rightmost point in the hull and
    // working anti-clockwise, we determine the first point that forms a convex angle in a
    // clockwise direction.
    $a = i - 1$
    while not Convex$(p[p[a].anti].pt,\ p[a].pt,\ p[i].pt)$
        $a = p[a].anti$
    // Put in the connections to new point i.
    LinkClockwise$(i, c)$; LinkClockwise$(a, i)$
```

The problem is to show that the second phase of adding all the points (described above) is $O(n)$ using amortised analysis.

Sample solution. We want to show the whole of the second phase is $O(n)$ complexity. However, the **for** loop is executed $O(n)$ times, and each iteration contains two nested **while** loops each of which is $O(n)$ complexity in the worst case. To show that the whole of the second phase is $O(n)$ we can show that each iteration has $O(1)$ amortised complexity.

As our measure of time we count the number of calls on the function CONVEX. The actual cost of one iteration of the **for** loop is then 2 plus the sum of the number of times, say k, that the bodies of the two **while** loops are executed. That is, the actual cost is $k + 2$.

We need to choose a potential function so that the difference in the cost of the execution of the **while** loops and the change in the potential function is a constant. We choose as our potential function the number of (bidirectional) links in the hull. This has the desired property that the potential function is always greater than or equal to the initial potential of zero.

On each iteration of the **for** loop, links are established from the new point (i) back to its clockwise (c) and anti-clockwise (a) neighbours in the hull. This constitutes an increase in potential of 2. Every time the body of a **while** loop is executed a link is effectively removed from the hull. Hence the decrease in the potential function due to the **while** loops is equal to the sum of the number of times the bodies of the two **while** loops are executed, i.e., k. The overall potential change for a single iteration of the **for** loop is thus $2 - k$. The amortised complexity is the sum of the actual complexity and the change in potential:

$$a = (k + 2) + (2 - k) = 4.$$

(For each point we add in two links (cost 2) and can potentially delete those two links later on (cost 2). So, each point contributes a cost of at most 4 to the computation of the convex hull.)

Hence the amortised complexity of the second phase of computing the convex hull is given by,

$$T(n) = 4 + 4 \times (n - 3),$$

giving a amortised complexity of $O(n)$. The initial constant 4 consists of a cost of 1 for the call to CONVEX in the initial **if** command, and a change of potential of 3 to set up the initial hull. As the potential function is always greater than the initial potential, this is also a bound on the actual complexity.