# COMP-251 Data Structures and Altorithms
# Exam #2 Solution

## 2. The Largest Gap Problem

### (a) Algorithm

The idea is to hash the $n$ numbers into $n+1$ buckets, using the floor function. Each bucket $i$ retains the largest value $LOW[i]$ and smallest value $HIGH[i]$ value in bucket. Finally we find the largest gap by finding the largest distance between the largest value of a bucket and the smallest value of the next non-empty bucket.

**Input:** $n$ real numbers $S[1:n]$ (unsorted)
**Output:** The largest gap, i.e. the length of the largest gap between consecutive numbers in sorted order.

**procedure** MaxGap

$MIN := min\ S[i]$;
$MAX := max\ S[i]$;
for $i := 1$ to $n+1$ do
begin
    $COUNT[i] := 0$;
    $LOW[i] := HIGH[i] := 0$;
end
for $i := 1$ to $n$ do
begin
    $index := 1 + \lfloor n \times (S[i] - MIN) \div (MAX - MIN) \rfloor$;
    $COUNT[index] := COUNT[index] + 1$;
    $LOW[index] := min(S[i], LOW[index])$;
    $HIGH[index] := max(S[i], HIGH[index])$;
end
$MAXGAP := 0$;
$LEFT := HIGH[i]$;
for $i := 2$ to $n+1$ do
begin
    if $COUNT[i] \neq 0$ then
    begin
        $THISGAP := LOW[i] - LEFT$;
        $MAXGAP := max(THISGAP, MAXGAP)$;
        $LEFT := HIGH[i]$;
    end
end
return $MAXGAP$

### (b) Correctness

Since $n$ numbers have been placed in $n+1$ buckets, so by the pigeonhole principle some bucket must be empty. This means that the largest gap cannot occur between two points in the same bucket. Thus, the largest gap must be between the largest value of a bucket and the smallest value of the next non-empty bucket.

### (c) Complexity

Find $MAX$ and $MIN$: O(n)
Create and initialize the buckets: O(n)
Put the numbers into the buckets and update COUNT, LOW, HIGH: O(n)
A single pass through the buckets to find the largest gap: O(n)

## 3. Dynamic Programming

### (a) Algorithm

Refer to
*http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/*
for detailed information.

The table should be like this:

| | m | a | t | e | r |
|---|---|---|---|---|---|
| m | 0 | 1 | 2 | 3 | 4 |
| u | 1 | 1 | 2 | 3 | 4 |
| s | 2 | 2 | 2 | 3 | 4 |
| t | 3 | 3 | 2 | 3 | 4 |

$\rightarrow$: insertion, $\downarrow$: deletion, $\searrow$: substitution

The edit-distance is 4, and you can find it in the right-bottom block.

### (b) Complexity

Suppose the first string has a length of $m$ and the second one has length $n$, then you need $O(m \cdot n)$ to fill up the $m \times n$ table.

### (c) Minimum-cost Path

Trace back from the right-bottom block to find the optimal path, i.e., find the previous block that optimally provokes the current value. For example, assume the right-bottom block has a coordinate $(4, 5)$, then this value 4 must be from $(4, 4)$ by an insertion or $(3, 4)$ by a substitution. A deletion from $(3, 5)$ will make the value 5, and thus is not optimal. We do so until reaching the left-top block. Note that there can be more than one minimum-cost paths.
Here is one minimum-cost path: $(0, 0) \searrow (1, 1) \rightarrow (1, 2) \searrow (2, 3) \searrow (3, 4) \searrow (4, 5)$
So the converting sequence is: must, maust, matst, matet, mater