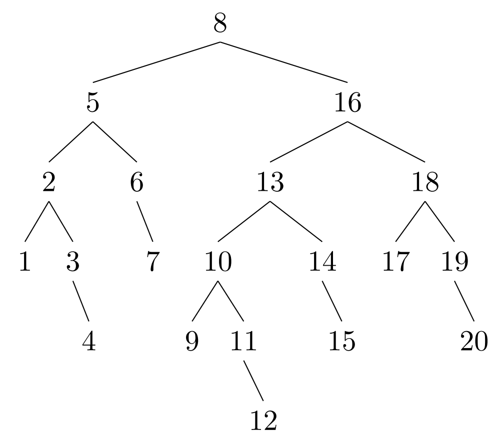
THE UNIVERSITY OF MELBOURNE
SCHOOL OF COMPUTING AND INFORMATION SYSTEMS
COMP90038 ALGORITHMS AND COMPLEXITY

Assignment 2, Semester 2, 2017

Student Name: XX
Student No.: XXXXXX

My Answers

Challenge 1 **(1 mark)**



Challenge 2 **(2 mark)**

Maybe the Right Answer: $d_{worst} = 5$, drops in sequence: $5, 9, 12, 14$.
Wrong Answer: According to the following strategy, in the worst case, $d = 6$.

Now Dr. Luator have 2 detectors to use in the test, she could start the test from not level 1 but level 4. After the dropping the first detector, if it's safe, she should shift 4 levels and drop on level 8, if safe once again, shift to and drop on level 12, and so forth. When she reaches level 15 and there's no more room for a 4-level-shifting, Dr. Luator should drop the first detector on level 15. If the first detector remains safe in all these tests, then level 15 is the safe limit.

If the first detector breaks in any of the above tests, for example, on level 4, then she should drop the second detector in the brute force way from level 1 to level 3. Similarly, if the first detector, say, breaks on level 12, Dr. Luator should drop the second one in brute force way, starting from level 9 until level 11. The reason she starts from level 9 is that she already tested on level 8 and the first detector is safe, so all the levels before level 8 is ensured to be safe. If the second detector remains safe in the brute force way test, then the safe limit should be the latest drop of the first detector. If the second detector breaks, then the safe limit is the last drop of the second detector.

When using this startegy, in the worst case, the safe limit floor n should be $10, 11, 13, or 14$, and the drop d should be 6.

Challenge 3 **(3 mark)**

1.
```
function F1(A[·], B[·], n, s)
    //Implement the worst-case running time of Θ(n^2)
    //solution to the problem
    //Input: Sorted arrays A[·], B[·] each with n positive
    //integer keys, and a positive integer s
    //Output: The pair of indices (i, j) if A[i] + B[j] = s or
    //(0, 0) if no such pair exists
    for i ← 1 to n do
        for j ← 1 to n do
            if A[i] + B[j] = s then
                return (i, j)
    return (0, 0)
```

2.
```
function F2(A[·], B[·], n, s)
    //Implement the worst-case running time of Θ(n log n)
    //solution to the problem
    //Input: Sorted arrays A[·], B[·] each with n positive
    //integer keys, and a positive integer s
    //Output: The pair of indices (i, j) if A[i] + B[j] = s or
    //(0, 0) if no such pair exists
    for i ← 1 to n do                      ▷ Linear scan on the array A[·]
        lo ← 1
        hi ← n
        while lo ≤ hi do
            j ← ⌊(lo + hi)/2⌋              ▷ Binary search on the array B[·]
```

```
            if B[j] = s − A[i] then
                return (i, j)
            if B[j] > s − A[i] then
                hi ← j − 1
            else
                lo ← j + 1
    return (0, 0)
```

3.
```
function F3(A[·], B[·], n, s)
    //Implement the worst-case running time of Θ(n)
    //solution to the problem
    //Input: Sorted arrays A[·], B[·] each with n positive
    //integer keys, and a positive integer s
    //Output: The pair of indices (i, j) if A[i] + B[j] = s or
    //(0, 0) if no such pair exists
    i ← 1
    j ← n
    while i ≤ n and j ≥ 1 do        ▷ Linear scans on the two arrays
        if A[i] + B[j] = s then
            return (i, j)
        if A[i] + B[j] < s then
            i ← i + 1        ▷ ignore all keys before A[i] and A[i] itself
        else
            j ← j − 1        ▷ ignore all keys after B[j] and B[j] itself
    return (0, 0)
```

Challenge 4 **(2 mark)**

1.
```
function CanScreen(U[·], V[·], d)
    //Implement the algorithm which determines, given
    //two vectors u and v, whether u screens v.
    //Assuming that vectors u and v are arrays
    //Input: The vectors as array U[·] and V[·] with d dimensions
    //Output: Returns True iff u screens v
    MergeSort(U[·])
    MergeSort(V[·])                        ▷ sort the arrays
    i ← 1
```

```
    j ← 1
    while i ≤ d and j ≤ d do        ▷ Linear scans on the two arrays
        if U[i] ≤ V[j] then
            return False
        i ← i + 1
        j ← j + 1
    return True
```

Complexity Analysis: The worst case running time of Mergesort is $\Theta(n \log n)$. The comparisons of each items in the two arrays requires the linear scans on the two arrays, of which the worst-case running time is $\Theta(n)$. Hence, the worst-case time complexity of the algorithm is $C(n) \in O(2n \log n + 2n) = O(n \log n)$.

Challenge 5 **(2 mark)**

1.

```
function Merge(M1, M2) : MultiwayTree
    if M2 = void then
        return M1
    if M1 = void then
        return M2
    M ← new MultiwayTree
    if M1.key > M2.key then
        M.key ← M1.key
        M.subtrees ← AddToList(M2, M1.subtrees)
    else
        M.key ← M2.key
        M.subtrees ← AddToList(M1, M2.subtrees)
    return M
```

2.

```
function MergePairs(L) : MultiwayTree
    if L = null then
        return void
    M1 ← L.elt
    L1 ← L.next
    if L1 = null then
        return M1
    M2 ← L1.elt
    L2 ← L1.next
```

```
    return Merge(Merge(M_1, M_2), MergePairs(L_2))
```

3.
```
function AddToList(M, L) : TreeList
    R ← new TreeList
    R.elt ← M
    R.next ← L
    return R
```

4.
```
function DeleteMax(M) : MultiwayTree
    if M = void then
        Error("Cannot delete from empty tree")
    return MergePairs(M.subtrees)
```

5.
```
function Insert(x, M) : MultiwayTree
    M′ ← new MultiwayTree
    M′.key ← x
    M′.subtrees ← null
    return Merge(M′, M)
```