The University of Melbourne
School of Computing and Information Systems
COMP90038 Algorithms and Complexity

Assignment 2, Semester 2, 2019

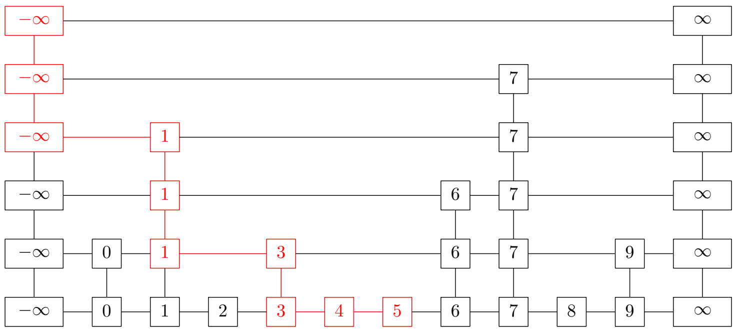
Sample Solution

Objectives

To improve your understanding of data structures and algorithms for sorting and search. To consolidate your knowledge of trees and tree-based algorithms. To develop problem-solving and design skills. To develop skills in analysis and formal reasoning about complex concepts. To improve written communication skills; in particular the ability to use pseudo-code and present algorithms clearly, precisely and unambiguously.

Problems

1. [5 Marks] A *skip list* is a randomised binary search structure, which has many of the desirable properties of a balanced tree. It is a sorted linked list with random shortcuts, as illustrated in the figure below. Search in a single-linked list of length n takes $\mathcal{O}(n)$. To speed it up, we construct another list containing about half of the items from the original, i.e., first, each item in the original list is duplicated with probability 1/2; then, all the duplicates are linked up; finally, we add a pointer from each duplicate back to the original. Sentinels are placed at the beginning and at the end of each list. We repeat recursively until the last sorted list contains sentinels only.



The search algorithm is presented below. We assume that each node has the fields $\{key, up, down, left, right\}$, where key represents the key value, and the remaining four are pointers to the adjacent nodes or NULL. Starting at the leftmost node L in the highest level, we scan through each level as far as we can without passing the target value x, and then proceed down to the next level. The search ends when we find the node with the largest key that is less or equal than x. Since each level of the skip list has about half of the nodes as the previous, the total number of levels is $\mathcal{O}(\log n)$, and the overall expected search time is $\mathcal{O}(\log n)$.

```
function SEARCH($x, L$)
    $u \leftarrow L$
    while $u.down \neq$ NULL do
        $u \leftarrow u.down$                    ▷ Drop a level down
        while $x \geq u.right.key$ do
            $u \leftarrow u.right$                ▷ Scan forward
    return $u$
```

(a) [2 Marks] Unscramble the routine INSERTNODE (x, u), which inserts a node with key x after u. This routine uses a function COINFLIP () that returns TRUE or FALSE with 1/2 probability, a function NEWNODE (x, u, v) which creates a new node with x as key after u and above v, and a function ISHEAD (u) that tests whether the u is the head of the list.

```
$u \leftarrow u.up$
while COINFLIP ()
$v \leftarrow$ NEWNODE $(x, u, v)$
if ISHEAD $(u)$ then
$u \leftarrow u.left$
function INSERTNODE $(x, u)$
$v \leftarrow$ NEWNODE $(x, u,$ NULL$)$
while $u.up =$ NULL and $u.left \neq$ NULL do
$w \leftarrow$ NEWNODE $(-\infty,$ NULL$, u)$
NEWNODE $(\infty, w, v.right)$
return $v$
```

R/
This algorithm is created in such way that there is always a "sentinel only" level, which is the reason there is no need to keep track of the height of the structure. The algorithm has $O(1)$ complexity, due to the COINFLIP () function.

```
function INSERTNODE($x, u$)
    $v \leftarrow$ NEWNODE $(x, u,$ NULL$)$
    while COINFLIP () do
        while $u.up =$ NULL and $u.left \neq$ NULL do
            $u \leftarrow u.left$                          ▷ Scan backwards
        $u \leftarrow u.up$
        $v \leftarrow$ NEWNODE $(x, u, v)$ ▷ Add a position to the tower of the new entry
        if ISHEAD $(u)$ then
            $w \leftarrow$ NEWNODE $(-\infty,$ NULL$, u)$     ▷ Place a sentinel above the head
            NEWNODE $(\infty, w, v.right)$              ▷ Place a sentinel above the tail
    return $v$
```

(b) [3 Marks] Now suppose that you are given two skip lists, one storing a set A of **unique** m keys, and another a set B of **unique** n keys, with $m > n$. There are **no repeats** between lists. Design a routine called MERGE (A, m, B, n) that merges B into A, storing the set $A \cup B$ in $\mathcal{O}(m + n)$ expected time. **Do not assume that every key in A is smaller than every key in B.** This routine should have about 15 lines of code.

R/
There are many ways to implement this method, but in essence the first step is to reach the bottom level of both lists. Then, the first valid key of B needs to be recovered. The next step is to linearly scan A until we find the location for the current key from B. Using the InsertNode $(\cdot, \cdot)$, the keys from B are added until there are no more keys left. Ideally, the routine should return the new head of A. While this implementation does not use m and n, other implementations may use them to stop the **while** loops.

```
1: function Merge($A, B$)
2:     $u \leftarrow$ Search $(-\infty, B)$                ▷ Find the lowest level of $B$.
3:     $u \leftarrow u.right$         ▷ This is the first element of $B$ with a valid key.
4:     $v \leftarrow$ Search $(-\infty, A)$                ▷ Find the lowest level of $A$.
5:     while $u.key \neq \infty$ do              ▷ Are there more keys in $B$ to add?
6:         while $u.key > v.right.key$ do
7:             $v \leftarrow v.right$                               ▷ Scan forward
8:         InsertNode $(u.key, v)$                          ▷ Insert the node.
9:         $u \leftarrow u.right$              ▷ Move to the next key in $B$ to insert in $A$.
10:    while $A.up \neq$ NULL do
11:        $A \leftarrow A.up$
12:    return $A$
```

2. [4 Marks] Consider two sets of integers, $X = [x_1,\ x_2,\ \ldots,\ x_n]$ and $Y = [y_1,\ y_2,\ \ldots,\ y_n]$. Write two versions of a FINDUNCOMMON(X, Y) algorithm to find the uncommon elements in both sets. Each of your algorithms should return an array with the uncommon elements, or an empty array if there are no uncommon elements.

   You may make use of any algorithm introduced in the lectures to help you develop your solution. That is, you do not have to write the 'standard' algorithms – just use them. Therefore, you should be able to write each algorithm in about 10 lines of code. **You must include appropriate comments in your pseudocode.**

   (a) [2 Marks] Write a pre-sorting based algorithm of FINDUNCOMMON(X, Y). Your algorithm should strictly run in $\mathcal{O}(n \log n)$.

   R/
   The approach presented below will result in the set not being sorted.

```
function FINDUNCOMMON($X, Y$)
    $X \leftarrow$ MERGESORT($X$)                              ▷ Sorting is $\mathcal{O}(n \log n)$
    $Y \leftarrow$ MERGESORT($Y$)                              ▷ Sorting is $\mathcal{O}(n \log n)$
    $j \leftarrow 0$
    for $i \leftarrow 0$ to $n - 1$ do                ▷ The complete loop is $\mathcal{O}(n \log n)$
        if BINSEARCH($X, Y[i]$) = FALSE then
            $Z[j] \leftarrow Y[i]$
            $j \leftarrow j + 1$
        if BINSEARCH($Y, X[i]$) = FALSE then
            $Z[j] \leftarrow X[i]$
            $j \leftarrow j + 1$
    return $Z$
```

   (b) [2 Marks] Write a Hashing based algorithm of FINDUNCOMMON(X, Y). Your algorithm should run in $\mathcal{O}(n)$.

   R/
   Assuming that the Hash table is correctly implemented, both loops are linear scans that should be completed in $\mathcal{O}(n)$.

```
function FINDUNCOMMON($X, Y$)
    $H_1 \leftarrow$ INITIALIZEHASHTABLE()
    $H_2 \leftarrow$ INITIALIZEHASHTABLE()
    for $i \leftarrow 0$ to $n - 1$ do                      ▷ The complete loop is $\mathcal{O}(n)$
        INSERT($H_1, X[i]$)
        INSERT($H_2, Y[i]$)
    $j \leftarrow 0$
    for $i \leftarrow 0$ to $n - 1$ do                      ▷ The complete loop is $\mathcal{O}(n)$
        if SEARCH($H_2, X[i]$) = FALSE then
            $Z[j] \leftarrow X[i]$
            $j \leftarrow j + 1$
        if SEARCH($H_1, Y[i]$) = FALSE then
            $Z[j] \leftarrow Y[i]$
            $j \leftarrow j + 1$
    return $Z$
```

3. [6 Marks] Klutzy Pty Ltd's CEO has asked you to help organise its end of the year *splurge*. Everybody invited, **one of which must be the CEO**, has to attend. Employees at Klutzy are organised into a strict hierarchical structure resembling a binary tree, where the root node represents the CEO. The data scientists at human resources have assigned to each employee an *awkwardness* score, α. An $\alpha > 0$ indicates that the staff member and its supervisor dislike each other, whereas an $\alpha < 0$ indicates that they actually like each other. If the guest list does not include an employee and its supervisor, then the added awkwardness is zero. To help you model Klutzy Pty Ltd assume that each node has the following fields: (a) L is a pointer to the left child or NULL if there is no left child; (b) R is a pointer to the right child or NULL if there is no right child; and (c) *alpha* is the awkwardness score of the inviting the employee and its supervisor. Provide a method that finds the score of the least awkward party, i.e., minimise the overall awkwardness score. For example, your algorithm should return '-1' and '0' for the two trees in the figure below. You must provide:

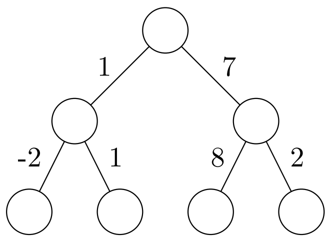
   (a) [2 Marks] A description of the optimisation problem to be solved, including an equation of the recursive relationship.

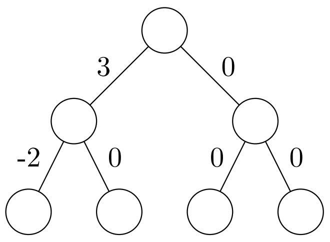
   R/
   Let us define $k_\uparrow$ as a binary variable that determines whether the supervisor of T was invited or not, and k as a binary variable that determines whether T was invited or not. Then our cost function is defined as:

   $$f(T, k_\uparrow) = \min_k \{f(L, k) + f(R, k) + \alpha k_\uparrow k\}$$

   This function would raise the following cases for each employee: (I) T does not exist; (II) L exists and T is invited and its supervisor was not; (III) L exists and T and its supervisor are invited; (IV) L exists and T is not invited; (V) R exists and T is invited and its supervisor was not; (VI) R exists and T and its supervisor are invited; and (VII) R exists and T is not invited. When T is not invited it does not matter whether its supervisor was or not invited as $\alpha k_\uparrow k = 0$. If T does not exist, then $f(\text{NULL}, k_\uparrow) = 0$. This is the basis case for the recursion. The CEO is a special case, as it must be invited to the party and has no supervisor. Hence $k_\uparrow = 0$ and $k = 1$.

   (b) [4 Marks] The pseudo-code of your method, in about 15 lines of code.





R/

The method consists of two routines. The first calculates the awkwardness of a generic tree. The second one restricts the result considering that the CEO must be invited to the party. Using the memoing approach below, the method has complexity $\mathcal{O}(n)$, as each node is visited at most twice. An approach without memoing is shorter and acceptable but slower. It is also allowed to use a table to record the data. Such table should have as many rows as nodes, and four columns. However, dealing with non-existing nodes makes the whole solution much more complicated and space inefficient.

```
function FindTreeAwkwardness($T, k$)
    if $T$ = NULL then
        return 0
    if $T.L0$ = NULL then            ▷ If these fields are null, they do not exist yet
        $T.L0 \leftarrow$ FindTreeAwkwardness ($T.L$, FALSE)
    if $T.L1$ = NULL then
        $T.L1 \leftarrow$ FindTreeAwkwardness ($T.L$, TRUE)
    if $T.R0$ = NULL then
        $T.R0 \leftarrow$ FindTreeAwkwardness ($T.R$, FALSE)
    if $T.R1$ = NULL then
        $T.R1 \leftarrow$ FindTreeAwkwardness ($T.R$, TRUE)
    $A0 \leftarrow T.L0 + T.R0$
    $A1 \leftarrow T.L1 + T.R1 + (T.alpha \times k)$
    return $\min(A0, A1)$
```

```
function FindPartyAwkwardness($T$)
    $L1 \leftarrow$ FindTreeAwkwardness ($T.L$, TRUE)
    $R1 \leftarrow$ FindTreeAwkwardness ($T.R$, TRUE)
    return $L1 + R1$
```

Submission and Evaluation

- You must submit a PDF document via the LMS. Note: handwritten, scanned images, and/or Microsoft Word submissions are not acceptable — if you use Word, create a PDF version for submission.
- Marks are primarily allocated for correctness, but elegance of algorithms and how clearly you communicate your thinking will also be taken into account. Where indicated, the complexity of algorithms also matters.
- We expect your work to be neat — parts of your submission that are difficult to read or decipher will be deemed incorrect. Make sure that you have enough time towards the end of the assignment to present your solutions carefully. Time you put in early will usually turn out to be more productive than a last-minute effort.
- Number of lines are given as an **indication only** and should not be considered as the actual length of the code. Correct solutions could have a few lines more or less depending on your notation, but not many more (if you see yourself writing ten more extra lines, then you are probably doing something wrong).
- You are reminded that your submission for this assignment is to be your own individual work. For many students, discussions with friends will form a natural part of the undertaking of the assignment work. However, it is still an individual task. You should not share your answers (even draft solutions) with other students. Do not post solutions (or even partial solutions) on social media or the discussion board. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

  Please see `https://academicintegrity.unimelb.edu.au`

If you have any questions, you are welcome to post them on the LMS discussion board *so long as you do not reveal details about your own solutions.* You can also email the Head Tutor, Lianglu Pan (lianglu.pan@unimelb.edu.au) or the Lecturer, Andres Munoz-Acosta (munoz.m@unimelb.edu.au). In your message, make sure you include COMP90038 in the subject line. In the body of your message, include a precise description of the problem.

Late Submission and Extension

Late submission will be possible, but a penalty will apply: a flag-fall of 2 marks, and then 1 mark per 12 hours late. Extensions will only be awarded in extreme/emergency cases, assuming appropriate documentation is provided. Simply submitting a medical certificate on the due date will not result in an extension.