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

Assignment 2, Semester 1 2021

Sample Solutions

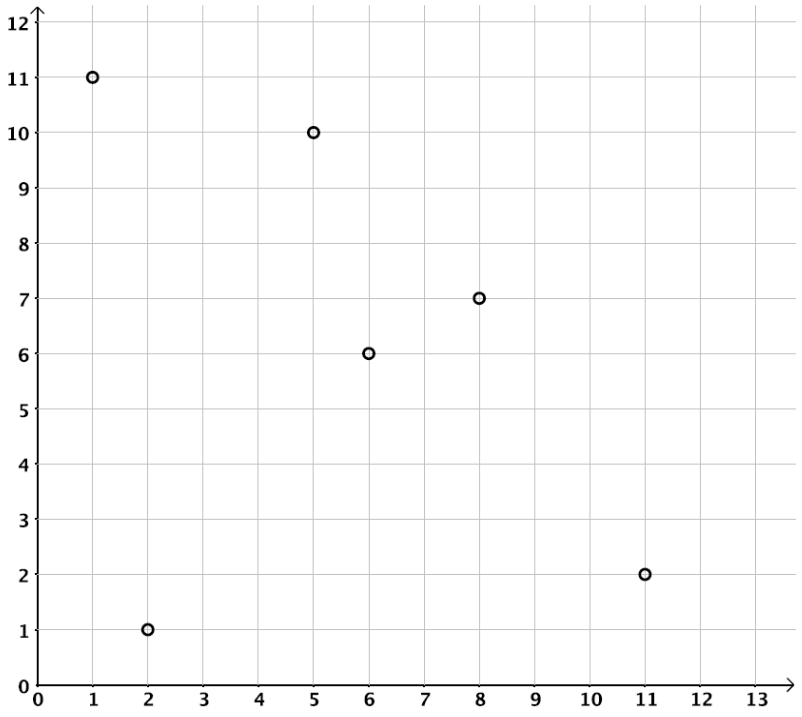
Objectives

To improve your understanding of the time complexity of algorithms and recurrence relations. To develop problem-solving and design skills. To improve written communication skills; in particular the ability to present algorithms clearly, precisely and unambiguously.

Problems

1. **[10 marks]**

   Let S be a finite set of n distinct points in $\mathbb{R}^2$. We say that a point $P = (x, y) \in S$ *governs* another point $P' = (x', y') \in S$ iff $x' \leq x$ and $y' \leq y$. A point is called *pareto optimal* for S if it is not governed by any other point in the set S. Write pseudocode for an algorithm that prints all the *pareto optimal* points of S. Your algorithm must run in $\mathcal{O}(nlogn)$ time for full marks.
   You may assume that the input are two coordinate arrays X and Y. For example, with the input $X = [11, 5, 6, 1, 8, 2]$ and $Y = [2, 10, 6, 11, 7, 1]$, your algorithm should print $(5, 10)$, $(1, 11)$, $(11, 2)$ and $(8, 7)$ (not necessarily in that order).



Solution:
Use a stable, $\mathcal{O}(nlogn)$ sorting algorithm (such as MERGESORT) to sort all nodes in descending order, twice: first by their y coordinates then by their x coordinates. Now the nodes are sorted in reverse x coordinate order, where nodes with the same x value are sorted in reverse y coordinate order.
To find pareto optimal points, we can process each node one-by-one, and keep track of the largest y value we've seen so far (store it in a variable, say, y_max). A node is pareto optimal if it has y value larger than the recorded y_max. We can observe that such a node is not governed by other node because all nodes we have processed so far have their y value smaller than this node, and all nodes we will process will either have smaller x values, or have same x value but smaller y value.
One possible algorithm is given below. The implementation of the MERGESERT function can be found in Assignment 1 sample solution.

```
function ParetoOptimal($X[0..n-1]$, $Y[0..n-1]$)
    Mergesort(Y, X)            ▷ Sort Y in descending order, update $X$ accordingly
    Mergesort(X, Y)            ▷ Sort X in descending order, update $Y$ accordingly
    $y\_max \leftarrow Y[0] - 1$
    for $i \leftarrow 0$ to $n - 1$ do
        if $Y[i] > y\_max$ then
            Print($X[i]$, $Y[i]$)
            $y\_max = Y[i]$
```

2. **[5 marks]**

You are a hacker that recently got hired by a cybersecurity company. The team needs to be preemptive against any adversarial attacks. Your role is to develop such attacks so the defense team can prepare mechanisms to avoid them.

In this problem, you were able to inject code into the algorithm for inserting an element into a Binary Search Tree. Your goal is to **force the BST to always degrade to a "stick" (to the right), or more specifically, a linked list**.

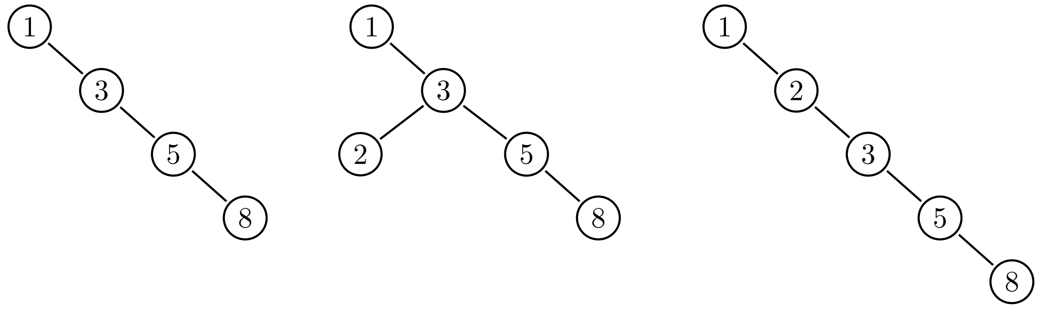
The algorithm to insert an element into the BST is as follows.

```
function BSTInsert($root$, $new$)
    if $new.value < root.value$ then
        if $root.left$ = NULL then
            $root.left \leftarrow new$
            Stickify($new$, $root$)
        else
            BSTInsert($root.left$, $new$)
    if $new.value > root.value$ then
        if $root.right$ = NULL then
            $root.right \leftarrow new$
            Stickify($new$, $root$)
        else
            BSTInsert($root.right$, $new$)
```

Here $root$ and new are nodes with fields $left$ and $right$ (which are pointers to other nodes) and a field $value$.

If the BST is already a "right stick", then running the BSTINSERT algorithm (with your STICKIFY function) should always leave the tree as a "right stick".

For example if we start with the tree on the left and insert 2 we will get the tree in the middle. After STICKIFY is run, we should get the tree on the right.



Your task is to write the pseudocode for the algorithm STICKIFY which takes the newly inserted node along with its parent and performs any necessary rotations to ensure the tree remains a "right stick". You may use ROTATERIGHT(*node*) and ROTATELEFT(*node*) without implementing these functions. In the above example, the ROTATERIGHT(3) should be called.

Solution:
There are two cases after inserting a node:

(a) The new node is inserted as a right child. No rotation needed in this case.

(b) The new node is inserted as a left child. A right rotation needs to be performed.

```
function STICKIFY($new$, $root$)
    if $root.left == new$ then
        ROTATERIGHT($root$)
```

3. **[5 marks]**

Consider the following hybrid sorting algorithm which runs standard MERGESORT but when one of the subarrays reach size 10 or smaller, it calls INSERTIONSORT on that array:

```
function HYBRIDSORT($A[0..n-1]$)
    if $n > 10$ then
        $B[0..\lfloor n/2 \rfloor - 1] \leftarrow A[0..\lfloor n/2 \rfloor - 1]$
        $C[0..\lceil n/2 \rceil - 1] \leftarrow A[\lfloor n/2 \rfloor..n-1]$
        HYBRIDSORT($B[0..\lfloor n/2 \rfloor - 1]$)
        HYBRIDSORT($C[0..\lceil n/2 \rceil - 1]$)
        MERGE($B[0..\lfloor n/2 \rfloor - 1], C[0..\lceil n/2 \rceil - 1], A[0..n-1]$)
    else
        INSERTIONSORT($A[0..n-1]$)
```

What is the worst case time complexity of HYBRIDSORT in terms of n? Justify your answer.

Solution:
The algorithm has a complexity of $\Theta(nlogn)$.
The number of operation required for a given input n is:

$$T(n) = \begin{cases} 2T(n/2) + f(n) & n > 10 \\ T_{InsertionSort}(n) & n \leq 10 \end{cases}$$

where $f(n)$ is the number of other operations including the Merge function and creating arrays B and C, which is in $\Theta(n)$.
The InsertionSort algorithm has a time complexity of $\Theta(n^2)$. However, the InsertionSort algorithm is only called when n is no more than 10 (a constant). Sort a constant number of elements using InsertionSort takes constant time.
Master theorem can be used here to solve this recurrence relation.

4. **[10 marks]**

Consider a two-player game. An array is given, containing n non-negative integers. Each player will take turn, to take one integer from either the beginning or end of the remaining array. This process is repeated until the array becomes empty. For both players, the goal is to maximise the sum of integers taken. Assume that both player will take their best move at each step.

(a) Provide an equation of the recursive relationship, for maximising the sum of integers taken. You may optionally explain it in a few sentences.

(b) Based on this recursive relationship, write the algorithm FirstPlayerSum($A[0..n-1]$) to calculate the sum of integers taken by the first player.

(c) State the time complexity of your algorithm.

Solution:

$$T(A[i..j]) = \begin{cases} max(A[i] + sum(A[i+1..j]) - T(A[i+1..j]), \\ \qquad A[j] + sum(A[i..j-1]) - T(A[i..j-1])) & i < j \\ A[i] & i = j \end{cases}$$

$T(A[i..j])$ describes the maximal sum of the integers taken by the next player. At each step, a player can take either $A[i]$ or $A[j]$ from the remaining array. If the player chose to take $A[i]$, the other player will need to solve the sub-problem with input $A[i+1..j]$, leaving $sum(A[i+1..j]) - T(A[i+1..j])$ for the current player.

```
function FirstPlayerSum($A[0..n-1]$)
    init $D[0..n-1][0..n-1] \leftarrow [[-1,-1,...,-1],[-1,-1,...,-1],...,[-1,-1,...,-1]]$
                        ▷ Global, $D[i][j]$ stores the sum of integers taken by the next player
    init $S[0..n-1]$                    ▷ Global, $S[i]$ stores the sum of $A[0..i]$
    $S[0] \leftarrow A[0]$
    for $i \leftarrow 1$ to $n-1$ do
        $S[i] \leftarrow S[i-1] + A[i]$
    return FindMaxSum($A[0..n-1]$)
function FindMaxSum($A[i..j]$)
    if $D[i][j] \neq -1$ then
        return $D[i][j]$
    if $i = j$ then
        return $A[i]$
    $t1 \leftarrow A[i]$ + FindSum($i+1, j$) − FindMaxSum($A[i+1..j]$)
    $t2 \leftarrow A[j]$ + FindSum($i, j-1$) − FindMaxSum($A[i..j-1]$)
    $D[i][j] \leftarrow max(t1, t2)$
    return $D[i][j]$
function FindSum($i, j$)                ▷ Finds the sum of $A[i..j]$
    if $i = 0$ then
        return $S[j]$
    return $S[j] - S[i-1]$
```

The overall time complexity of this algorithm is $\Theta(n^2)$.
The initialisation of 2D array $D[.][.]$ takes $\Theta(n^2)$ time. The initialisation of array $S[.]$ takes $\Theta(n)$ time. The FindSum function takes constant time to return the result. The function FindMaxSum can be called twice for each distinct pair of (i, j), though the calculation will only be done once (the second time it's called, the result $D[i][j]$ is directly returned).