THE UNIVERSITY OF BRITISH COLUMBIA
CPSC 320: MIDTERM EXAMINATION – November 8, 2021

Full Name: ______________________ CS Ugrad ID: _______________

Signature: ______________________ UBC Student #: |__|__|__|__|__|__|__|__|

Important notes about this examination

1. You have 90 minutes to complete this exam.
2. ONE double-sided, letter-size note sheets are allowed.
3. All original pages of this booklet must be returned when you finish your exam.
4. Good luck!

1 236413 048929

This page intentionally left (almost) blank.
Do not write answers here; there is an extra page at the end.

CPSC 320 2021W1: Midterm Exam 2

November 8, 2021

1 ID Please (1 mark)

Write your CS undergraduate login ID again in this box:

2 Mystery Recurrence (11 marks)

An algorithm that solves a mystery problem has runtime described by the following recurrence, where $c > 0$:

$$T(n) = \begin{cases} c, & \text{for } n \leq 3, \\ 2T(n/3) + T(n/4) + cn^2, & \text{for } n > 3. \end{cases}$$

1. (3 marks) Draw or describe the first two levels (i.e., the root at level 0 and the nodes at level 1) of the recursion tree for this recurrence. Label each node with the problem size and the time to solve the mystery problem, not counting recursive calls.

   SOLUTION: Level 0 has one root node, labeled with size n and time cn^2.

   Level 1 has three nodes. Their labels are $(n/3, c(n/3)^2)$, $(n/3, c(n/3)^2)$, and $(n/4, c(n/4)^2)$.

2. (2 marks) What is the total time (not counting recursive calls) for nodes at level 1? No justification needed.

   SOLUTION:
   $$cn^2(1/9 + 1/9 + 1/16) = cn^2(2/9 + 1/16) = cn^2(41/144)$$

3. (2 marks) What is the total time (not counting recursive calls) for nodes at level i, assuming that there is no leaf at a level $\leq i$? No justification needed.

   SOLUTION: $cn^2(41/144)^i$

4. (3 marks) Which of these expressions is an *upper bound* on the runtime of the algorithm? **Check all that apply.**

   SOLUTION:

   [✓ $O(n^2)$] [✓ $O(n^2 \log n)$]

   [✓ $O(n^3)$] [✓ $O(3^n)$]

5. (3 marks) Which of these expressions is a *lower bound* on the runtime of the algorithm? **Check all that apply.**

   SOLUTION:

   [✓ $\Omega(n^2)$] $\Omega(n^2 \log n)$

   $\Omega(n^3)$ $\Omega(3^n)$

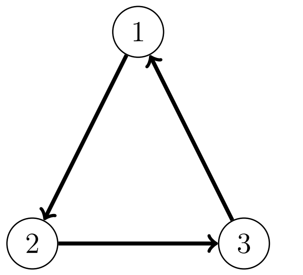
3 Finding a Place in the Pecking Order (16 marks)

This question continues the Pecking Order problem from the first midterm. To refresh your memory, here is the problem description from Midterm 1:

Small clusters of domesticated chickens typically maintain a pairwise pecking order: for *every* pair of chickens i and j, one dominates the other when it comes to pecking rights at food time.

A dominance relation can be represented by a *dominance graph* in which nodes represent chickens and a directed edge from i to j indicates that i dominates j. In a dominance graph, for every pair of distinct nodes i and j, there is a directed edge from i to j, or from j to i, but not both. A *pecking order* of $n \geq 1$ chickens is a permutation $L(1), \ldots, L(n)$ of the chickens such that for $1 \leq i \leq n-1$, chicken $L(i)$ dominates chicken $L(i+1)$.

For example, with three chickens where 1 dominates 2, 2 dominates 3, and 3 dominates 1, one pecking order is 1,2,3 and the dominance graph is



Midterm 1 gave an algorithm to compute a full pecking order of a dominance graph G with $n \geq 1$ nodes.

```
function PECKING-ORDER($G$)                    ▷ $G$ is a dominance graph with $n \geq 1$ nodes (chickens)
    $L \leftarrow (1)$                          ▷ $L$ is initialized to be the list containing chicken 1
    $i = 2$
    good $\leftarrow$ true
    while ($i \leq n$) and good do              ▷ try to insert chicken $i$ into the current pecking order $L$
        if $L(i-1)$ dominates $i$ then          ▷ $L(i-1)$ is currently the last item in list $L$
            insert $i$ at the end of the list $L$
        else if $i$ dominates $L(1)$ then       ▷ $L(1)$ is the first item in list $L$
            insert $i$ at the start of the list $L$
        else                                    ▷ $L(1)$ dominates $i$ and $i$ dominates $L(i-1)$
            good $\leftarrow$ false
            $k \leftarrow 1$
            while (not good) and ($k < i-1$) do
                if $L(k)$ dominates $i$ and $i$ dominates $L(k+1)$ then
                    insert $i$ between $L(k)$ and $L(k+1)$
                    good $\leftarrow$ true
                end if
                $k \leftarrow k+1$
            end while
        end if
        $i \leftarrow i+1$
    end while
    if good then
        return the list $L$
    else
        return “No full pecking order found”
```

```
27:     end if
28: end function
```

On **this** midterm, we will modify this algorithm slightly. In particular, we replace lines 11–19 of the above code with the following:

```
11:             $k \leftarrow$ FINDPECKINGPLACE($i, 1, i-1$)
12:             insert $i$ between $L(k)$ and $L(k+1)$
```

where FINDPECKINGPLACE is defined (except for the missing part that you need to figure out) as:

```
1: function FINDPECKINGPLACE($i$,low,high)
     // This function should compute an index $k$ with low $\le k <$ high such that
     // $L(k)$ dominates $i$, and $i$ dominates $L(k+1)$.
     // It should only be called with low < high, and when
     // $L$(low) dominates $i$, and $i$ dominates $L$(high).
2:     if low + 1 = high then
3:         return low                          ▷ Base Case: $k$ = low is a suitable index.
4:     else
            *** MISSING CODE ***
5:     end if
6: end function
```

Your job is to supply the "MISSING CODE" to make the function work correctly. For full credit, your solution must (1) work correctly, (2) run in $O(\log n)$ time, where $n =$ high $-$ low, and (3) be recursive. You are NOT allowed to modify anything else in the code. Write your answer below. You do not need to justify your code, but doing so may help you earn credit even if your code isn't sufficiently clear or correct.

SOLUTION: The key is to realize that you can do something that is essentially binary search. (The log time bound was a big hint, as well as the parameters being named low and high.) You check the midpoint between low and high. Call this m. If $L(m)$ dominates i, then you know that you can find a place to insert i somewhere after $L(m)$ and before L(high). On the other hand, if i dominates $L(m)$, then you know that you can find a place to insert i somewhere after L(low) and before $L(m)$.

This gives the following code:

```
 1: function FINDPECKINGPLACE($i$,low,high)
      // This function should compute an index $k$ with low $\le k <$ high such that
      // $L(k)$ dominates $i$, and $i$ dominates $L(k+1)$.
      // It should only be called with low < high, and when
      // $L$(low) dominates $i$, and $i$ dominates $L$(high).
 2:     if low + 1 = high then
 3:         return low                          ▷ Base Case: $k$ = low is a suitable index.
 4:     else
 5:         Let $m \leftarrow$ (low + high)/2
 6:         if $i$ dominates $L(m)$ then
 7:             return FINDPECKINGPLACE($i$,low,$m$)
 8:         else
 9:             return FINDPECKINGPLACE($i$,$m$,high)
10:         end if
11:     end if
12: end function
```

4 Study Plan: Part I (19 marks)

You have exams in C different courses, and have n total hours to prepare. Your grade on any exam depends on how many hours you spend studying for said exam. You have a 2D array $M[1..C][0..n]$, where for $1 \leq c \leq C$, $M[c][0]$ is the mark you'll get in course c if you don't study for it at all, and for $1 \leq j \leq n$, $M[c][j]$ records the additional points you'll get from the jth hour of study for course c. *Assume throughout this problem that for any given course, the additional marks gained from the jth hour of study is at least the marks gained from the $(j+1)$st hour of study.* That is,

$$M[c][j] \geq M[c][j+1], \text{ for any } 1 \leq c \leq C \text{ and } 1 \leq j < n.$$

Here's an example of such an array M, where $C = 3$ and $n = 5$.

M	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$c=1$	52	6	3	3	1	1
$c=2$	55	4	2	1	1	1
$c=3$	58	4	3	3	2	1

A *study plan*, represented as a list $h = (h_1, \ldots, h_C)$, describes how many hours of study you assign to each course. Each h_c is nonnegative, and $\sum_{c=1}^{C} h_c = n$. The marks you get if you use study plan h is

$$\begin{aligned} \text{marks}(h) &= \sum_{c=1}^{C}(M[c][0] + \ldots + M[c][h_c]) \\ &= \sum_{c=1}^{C}\sum_{j=0}^{h_c} M[c][j] \qquad \text{(if you prefer the sum written formally)} \end{aligned}$$

You want an *optimal* study plan, i.e., one that maximizes the sum of your course marks. For our example above, the study plan $h_1 = 2$, $h_2 = 2$, $h_3 = 1$ (i.e., 2 hours of study for course 1, 2 hours for course 2, and 1 hour for course 3) results in a suboptimal total of 184 marks:

$$(52 + 6 + 3) + (55 + 4 + 2) + (58 + 4) = 184.$$

1. (2 marks) For the above example (with $C = 3$ and $n = 5$), write down *two* optimal study plans. For each study plan, you should specify h_1, h_2, and h_3. You do not need to justify your answer.

   SOLUTION: One optimal study plan is $h_1 = 1$, $h_2 = 1$, and $h_3 = 3$. The total number of marks is then

   $$58 + 59 + 68 = 185.$$

   Another optimal study plan is $h_1 = 2$, $h_2 = 1$, and $h_3 = 2$; this also results in 185 marks.

2. (5 marks) Complete the following greedy algorithm so that it finds an optimal study study plan. A one-line description in the blank space below suffices.

   SOLUTION:

```
function StudyPlan($M[1..C][0..n]$)
    initially set $h_c$ to 0, for $1 \le c \le C$                  ▷ no hours allocated yet
    for $j$ from 1 to $n$ do                      ▷ decide how to allocate the $j$th hour of study
        ▷ describe here how to choose a course $c$ that gets an additional hour

        choose $c$ to be a course for which $M[c][h_c + 1]$ is maximal

        $h_c \leftarrow h_c + 1$                            ▷ allocate another hour to course $c$
    end for
    return $h = (h_1, h_2, \ldots, h_C)$
end function
```

   The intuition for the +1 in $M[c][h_c + 1]$ is that h_c indexes the last hour of study *you've already assigned* to course c, so $h_c + 1$ is the index of the value of the *next* hour of study. However, this was subtle enough that we didn't deduct any marks if you had written $M[c][h_c]$ instead of $M[c][h_c + 1]$. (Another alternative would have been if we had initialized the h_c to 1 instead of 0. The pseudocode would have been slightly cleaner, but stylistically, this isn't as good, because it breaks the definition that h_c is the number of additional hours that you study for course c, and makes it one more than the number of hours instead.)

3. (4 marks) What is the runtime of an efficient implementation of your greedy strategy? Justify your answer, briefly explaining what data structure is useful, in light of the strategy you use to choose a course at each iteration.

   SOLUTION: the runtime is $O(n \log C)$. There are n iterations of the **for** loop, and each one takes $O(\log C)$ time if a priority queue Q is used to keep track of the number of additional marks gained for each course c, when an additional hour is assigned to that course.

   [Details: Initially we create an empty queue Q, and insert each course c with key value $M[c][1]$. Then, to find the best course at each iteration j, we can use ExtractMax(Q) operation. If course c is extracted, then we can add one more line at the very end of the **for** loop, inserting c back into the queue, with key value $M[c][h_c + 1]$.]

4. (4 marks) Let h^G be the study plan produced by your greedy algorithm, and let h^* be an optimal study plan. Suppose that h^G and h^* are not identical. Then

   - For some course c, $h_c^G > h_c^*$ (greedy allocates more hours to course c than does h^*).
   - For some other course c', $h_{c'}^G < h_{c'}^*$ (greedy allocates fewer hours to c' than does h^*).

   Let h' be obtained from h^* by adding an hour to c, and removing an hour from c'. (So $h'_c = h_c^* + 1$, $h'_{c'} = h_{c'}^* - 1$, and $h'_i = h_i^*$ for all other courses i.) Show that

   $$\text{marks}(h') - \text{marks}(h^*) \geq 0.$$

   You can use without proof the following inequality (which should follow from your greedy strategy):

   $$M[c][h_c^* + 1] \geq M[c'][h_{c'}^*].$$

   SOLUTION: The study plans h' and h^* are identical except for two differences:

   - h' allocates $h_c^* + 1$ hours to course c, while S^* devotes only h_c^* hours. The extra marks that h' gets for course c is $M[c][h_c^* + 1]$.
   - h' allocates only $h_{c'}^* - 1$ hours to course c', while h^* devotes $h_{c'}^*$ hours. The extra marks that h^* gets for course c' is $M[c'][h_{c'}^*]$.

   Therefore,

   $$\text{marks}(h') - \text{marks}(h^*) = M[c][h_c^* + 1] - M[c'][h_{c'}^*] \geq 0,$$

   by the given inequality.

5. (4 marks) Using the previous part (and again assuming that the inequality given there holds for your greedy strategy and any optimal strategy h^*), explain why $\text{marks}(h^G) \geq \text{marks}(h^*)$.

   SOLUTION: Let h^G be the greedy strategy and h^* an optimal strategy. If these are not identical, then we modify h^* to get h' as in the previous part. Plan h' must be optimal, since the number of marks obtained by plan h' is at least the number obtained by h^*. Also, h' is "closer" to h^G since it allocates one more hour to course c and one less to c'. If h' is not identical to h^G we can repeat this "exchange" argument until they are identical, and all intermediate plans also must be optimal. Since we eventually reach h^G, it follows that h^G must be optimal.

5 Study Plan: Part II (14 marks)

In this question, we analyze **almost** the same problem as in the preceding Question **??** (Study Plan: Part I). We will use all the same definitions and notation (so go back and read Question **??** if you haven't already). **However, we make one important change: we eliminate the assumption that**

$$M[c][j] \geq M[c][j+1], \textbf{ for any } 1 \leq c \leq C \textbf{ and } 1 \leq j < n.$$

Again, we emphasize that for **this** question, you do **not** have the above assumption.

Therefore, for example, the following would be a legal array M (with $C = 3$ and $n = 5$) for **this** question, but not for Question **??**:

M	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$c=1$	52	1	1	3	3	7
$c=2$	55	4	2	1	1	1
$c=3$	58	4	1	3	2	3

As in Question **??**, a *study plan*, represented as a list $h = (h_1, \ldots, h_C)$, describes how many hours of study you assign to each course. Each h_c is nonnegative, and $\sum_{c=1}^{C} h_c = n$. The marks you get if you use study plan h is

$$\begin{aligned} \text{marks}(h) &= \sum_{c=1}^{C}(M[c][0] + \ldots + M[c][h_c]) \\ &= \sum_{c=1}^{C}\sum_{j=0}^{h_c} M[c][j] \qquad \text{(if you prefer the sum written formally)} \end{aligned}$$

You want an *optimal* study plan, i.e., one that maximizes the sum of your course marks. For our example above, the study plan $h_1 = 2$, $h_2 = 2$, $h_3 = 1$ (i.e., 2 hours of study for course 1, 2 hours for course 2, and 1 hour for course 3) results in a suboptimal total of 177 marks:

$$(52 + 1 + 1) + (55 + 4 + 2) + (58 + 4) = 177.$$

1. (1 mark) Give an optimal study plan for the example matrix above (with $C = 3$ and $n = 5$). For your answer, you should specify h_1, h_2, and h_3. You do not need to justify your answer.

   SOLUTION: The optimal solution is $h_1 = 5$, $h_2 = 0$, and $h_3 = 0$. This gives a total value of $(52 + 1 + 1 + 3 + 3 + 7) + 55 + 58 = 180$ (not a required part of your answer). Intuitively, the 7 that you get after 5 hours of study of course 1 is worth so much that it's worth sacrificing the other courses to get there.

2. (3 marks) Here's a recursive algorithm to compute the value (total marks) of an optimal study plan. For the remainder of this question, we will set $C = 3$, just to make the code easier to write.

   The key idea behind the algorithm is that if you have 0 hours of study time left, then you get whatever you have earned so far; at any other point in time, you have the choice of spending one more hour on any one of the 3 courses. In each of the 3 cases, the best you can do is to get the marks for that hour, plus the best possible score for your remaining study hours after that point in time. (We will assume the matrix M and the number of hours n are global variables, since our code has a lot of parameters already.)

```
1: function RecursiveStudyPlan($h_1$, $h_2$, $h_3$)    ▷ $h_1 \geq 0, h_2 \geq 0, h_3 \geq 0$, and $h_1 + h_2 + h_3 \leq n$
2:     if $n - h_1 - h_2 - h_3 = 0$ then              ▷ Base Case: If there are no hours left,
 3:         $a \leftarrow 0$                                   ▷ add up all the marks earned.
 4:         for $c = 1$ to 3 do
 5:             for $i = 0$ to $h_c$ do
 6:                 $a \leftarrow a + M[c][i]$
 7:             end for
 8:         end for
 9:         return $a$
10:     else            ▷ $v_i$ is the most marks we can get overall if we spend the next hour on course $i$
11:         $v_1 \leftarrow$ RecursiveStudyPlan$(h_1 + 1, h_2, h_3)$
12:         $v_2 \leftarrow$ RecursiveStudyPlan$(h_1, h_2 + 1, h_3)$
13:         $v_3 \leftarrow$ RecursiveStudyPlan$(h_1, h_2, h_3 + 1)$
14:         return $\max(v_1, v_2, v_3)$
15:     end if
16: end function
```

Calling RecursiveStudyPlan(0, 0, 0) computes the maximum number of marks you can earn with an optimal study plan. Note that the recursion is written somewhat strangely, with the number of hours spent going **up** on each recursive call. We coded things this way to work better with the notation we gave you. You can see that you won't get infinite recursion by noticing that the number of hours left is equal to $n - h_1 - h_2 - h_3$, which goes down by 1 in each recursive call.

Give a tight big-O bound in terms of n on the runtime when calling RecursiveStudyPlan(0, 0, 0). You do not need to justify your answer, but doing so might help you earn partial credit.

SOLUTION: $O(n3^n)$. The recursion always goes down n levels, with 3x more calls per level, which gives the 3^n factor. In the base case, the h_i always add up to n, so that gives a factor of n.

3. (10 marks) Memoize the recursive algorithm. You should call your main function MemoStudyPlan, and it should have the same interface as RecursiveStudyPlan, i.e., it should assume that the matrix M and the number of hours n are global variables, and that you call MemoStudyPlan(0,0,0) to compute the optimal number of marks. Your algorithm will memoize solutions in a 3D table, where soln$[h_1][h_2][h_3]$ stores the number of marks you earn when you spend h_1 hours on course 1, h_2 hours on course 2, and h_3 hours on course 3. You may also assume that your solution table is a global variable.

   You may write your code out entirely, or refer to the text of RecursiveStudyPlan with comments like "Insert lines 3–8 of RecursiveStudyPlan here."

   SOLUTION: Many minor variations in the code are OK.

```
 1: function MemoStudyPlan($h_1$, $h_2$, $h_3$)
 2:     for $i_1 = 0$ to $n$ do
 3:         for $i_2 = 0$ to $n - i_1$ do
 4:             for $i_3 = 0$ to $n - i_1 - i_2$ do
 5:                 soln$[i_1][i_2][i_3] \leftarrow -1$                 ▷ Flag for unknown values.
 6:             end for
 7:         end for
 8:     end for
 9:     return MSPHelper($h_1$,$h_2$,$h_3$)
10: end function
11: function MSPHelper($h_1$, $h_2$, $h_3$)
12:     if soln$[h_1][h_2][h_3] \neq -1$ then      ▷ Base case is expensive; better to check memo table first.
13:         return soln$[h_1][h_2][h_3]$
```

```
    else if $n - h_1 - h_2 - h_3 = 0$ then          ▷ Base Case: If there are no hours left,
        $a \leftarrow 0$                                ▷ add up all the marks earned.
        for $c = 1$ to 3 do
            for $i = 0$ to $h_c$ do
                $a \leftarrow a + M[c][i]$
            end for
        end for
        soln$[h_1][h_2][h_3] \leftarrow a$     ▷ If we check memo table for base case, must store result.
        return $a$
    else        ▷ $v_i$ is the most marks we can get overall if we spend the next hour on course $i$
        $v_1 \leftarrow$ MSPHelper$(h_1 + 1, h_2, h_3)$
        $v_2 \leftarrow$ MSPHelper$(h_1, h_2 + 1, h_3)$
        $v_3 \leftarrow$ MSPHelper$(h_1, h_2, h_3 + 1)$
        soln$[h_1][h_2][h_3] \leftarrow \max(v_1, v_2, v_3)$            ▷ Store the result in the memo table
        return soln$[h_1][h_2][h_3]$
    end if
end function
```

Extra page for scratch work

Extra page for scratch work

Extra page for scratch work