Assignment 1 Solutions

with tentative marking scheme

Dr. Wael Abouelsaadat

Course Weight:7.5% , **Score**: out of 395☺

Teams: to be done in teams of 3 (2< *3*< 4).

- ➔ Once Part A is submitted, you cannot change teams during course of assignment 1. If a team member is not participating, use the team member eval form that will be on MET/cms website, and individual evaluation sessions will be held.
- ➔ Part deadline is a hard one. Sample solution to parts submitted will be posted on MET/cms before midterm.
- ➔ Marks will be posted after complete submission of all parts of A1.
- ➔ Submission instructions will be posted by your TAs.
- ➔ Show your work! Even if you answer a problem incorrectly, you will be rewarded partial marks on how far you have progressed in your attempts towards the correct one.
- ➔ You can also present several solutions to the same problem.
- ➔ Follow the answers style presented in practice assignments for divide and conquer, greedy and dynamic programming.
- ➔ In each solution, start by writing your observations first if there is any, followed by brief high-level English description of your solution. Next, present the actual solution in pseudo-code implementation (pseudo code means it looks like code but not strictly following any programing language syntax – see https://en.wikipedia.org/wiki/Pseudocode).

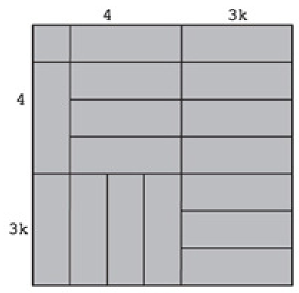
Part A: Divide and Conquer Algorithm Design

Assignment 1.1 [*30 marks*]

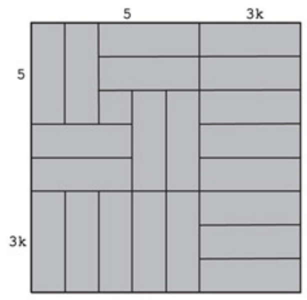
Imagine a marble slab of size 3m x 1m. One can tile any *n* x *n* room with such slab if *n* is divisible by 3. Is it true that for every *n* > 3 that is not divisible by 3, one can tile an *n* x *n* square with such slab and a single 1m x 1m slab? If it is possible, explain how, if it is not explain why. Use Divide and conquer in your answer.

Solution& marking

- *Observations & solution*:
  - Consider the case of *n* mod 3 = 1 and *n* > 3, that is *n* = 4 + 3*k* where *k* >= 0. We can divide the square into three sub regions: the 4 x square in, say, the upper left corner, the 4 x 3*k* rectangle, and 3*k* x (4 + 3*k*) rectangle (left below). The 4 x 4 square requires 1 slab placed in one of its corners to make it possible to tile the rest of it; tiling the other 2 rectangles (if *k* > 0) is trivial since both of them have a side equal to 3*k*.



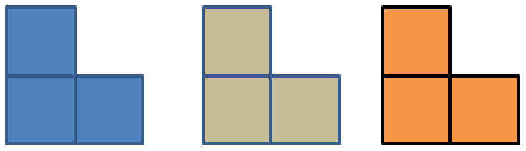
  - Similarly, if *n* mod 3 = 2 and *n* > 3, that is, *n* = 5 + 3*k* where *k*>= 0, we can tile the board as shown below.



- *Marking*:
  - Description, pseudo-code, code analogous to above gets full mark. Note that there were no algorithm required from students – just a proof that it works, so implementation details are unimportant.
  - Missing one of the two cases → -10
  - Solution that will miss filling a space in one of the two cases → -5
  - Major logical errors → -4
  - Minor logical errors → -2
  - Brute force approaches that fills from beginning to end and try to handle the unfilled space → *no marks*.
  - Not mentioning a split (the divide step) and proceeding from 2 or more corners to fill in until empty space is left in center is actually brute force → *no marks*.

Assignment 1.2 [*35 marks*]

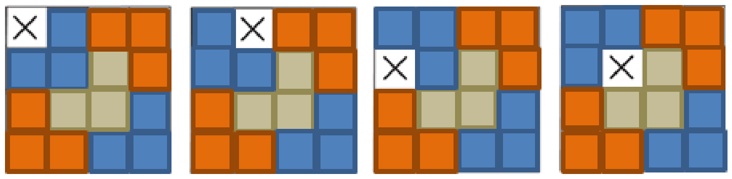
Imagine having a special type of ceramic that comes in an L-shape squares and in 3 distinct colors as shown below. Devise a divide and conquer solution for the following task: given a $2^n \times 2^n$ ($n > 1$) room with one missing square, tile it with this L-shaped ceramic so that no pair of ceramics that share an edge have the same color.



Solution& marking

- *Observations:*
  - This can be solved by the following algorithm if $n = 2$;
    - divide the board into four 2 x 2 boards and place one gray ceramic to cover the three central squares that not in the 2 x 2 board with the missing square.
    - Next, place one blue ceramic in the upper left 2 x 2 board, one orange ceramic in the upper right 2 x 2 board, one blue ceramic in the lower right 2 x 2 board, and one orange ceramic in the lower left 2 x 2 board.
  - For any location of the missing square, the following holds:
    - Going left to right, the upper edge of the board is covered by an alternating sequence of two blue squares, followed by two orange squares, with the missing square possibly replacing one square in this sequence.
    - Going down, the right edge of the board is covered by an alternating sequence of two orange squares, followed by two blue squares, with the missing square possibly replacing one square in this sequence.

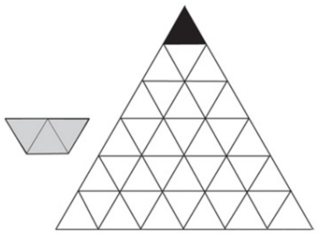
- Going right to left, the lower edge of the board is covered by an alternating sequence of two blue squares followed by two orange squares, with the missing square possibly replacing one square in this sequence.
- Going up, the left edge of the board is covered by an alternating sequence of two orange squares followed by two blue squares with the missing square possibly replacingone square in this sequence.



- *Algorithm:*
  - If n > 2, divide into four 2^{n}-1 x 2^{n}-1 board and place one gray ceramic to cover the 3 central squaresthat not in the 2^{n}-1 x 2^{n}-1 boards with the missing square. Then tile each of the three 2^{n}-1 x 2^{n}–1 boards recursively by the same algorithm.
- *Marking*:
  - Psudeo-code, code or description similar to above gets full mark
  - Solution that will miss filling a space in one of the two cases → -6
  - Major logical errors → -4
  - Minor logical errors → -2
  - Brute force approaches that fills from beginning to end and try to handle the unfilled space → *no marks*.
  - Not mentioning the split (the divide step) and proceeding from 2 or more corners to fill in until empty space is left in center is actually brute force → *no marks*.

Assignment 1.3 [35 *marks*]

An equilateral triangle is partitioned into a smaller equilateral triangles by parallel lines dividing each of its sides into $n > 1$ equal segments. The topmost equilateral triangle is chopped off yielding a region such as the one shown below for $n = 6$. This region needs to be tiled with trapezoid tiles made of three equilateral triangles of the same size as the small triangles composing the region. (Tiles need not be oriented the same way, but they need to cover the region exactly with no overlaps.) Determine all values of n for which this can be done and device a divide and conquer tiling solution for such n's.

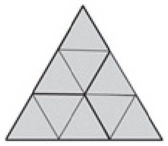


Solution& marking

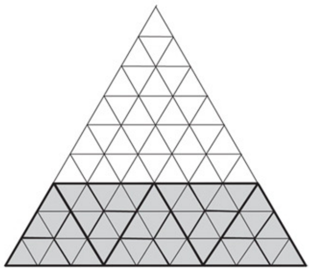
- *Observations:*
  - Counting the small triangles by the layers, starting with the base of the triangular region, weget:

$$T(n) = [\, n + 2(n-1) + 2(n-2) + \dots + 2 \times 1\,) - 1$$
$$= n + 2(n-1)n / 2 - 1 = n^2 - 1$$

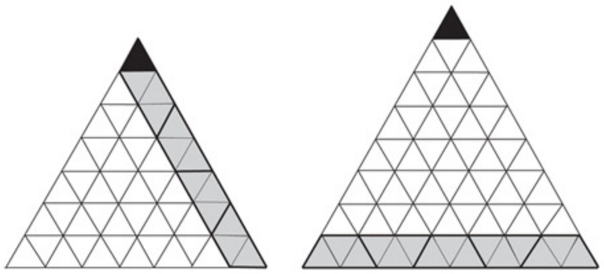
  - Since one tile is made of up of three small triangles, $n^2 - 1$ must be divisible by 3 for a tiling to exist. But, $n^2 - 1$ is divisible by 3 if and only if n is not divisible by 3, which immediately is established by considering the cases of $n = 3k$, $n = 3k + 1$, and $n = 3k + 2$.
  - If $n = 3k$ and no small triangle is removed from the region, it can be tiled by trapezoids.
    - For $k = 1$, region can be tiled by 3 trapezoids as shown below.



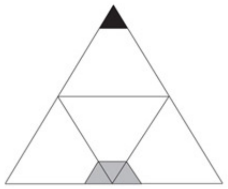
- If a trapezoid tiling exists for n = 3k where k>=1, it also exists for n = 3(k+1). This can be shown by considering the line parallel to the base through the points dividing the region's sides 3:3k as shown below. This line dissects the region into the trapezoid below this line and the equilateral triangle above the line. The former can be dissected into (k+1) + k equilateral triangles of side size 3 and hence by be tiled by trapezoid tiles; the later is already described above.



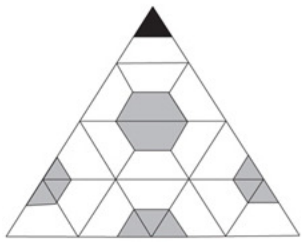
- If n = 3k + 2, we can place 2k + 1 trapezoids along the region's base to reduce this instance to that of n = 3k + 1, as shown below.



- *Algorithm for working n:*
  - *divide:* split the triangle by drawing 3 straight lines connecting the middle points of the three sides of the triangle as shown below.



- The lines will dissect the region into four congruent sub regions similar to the original region. Hence, each of them can be tiled by the same algorithm, that is recursively as shown below.



- *Conquer:* when we reach a region that we can fill directly without splitting, fill it.
- *Combine:* no work is done except putting all regions back together filled.

- *Marking*
    - Determining all values of n → 15 marks, algorithm → 20 marks
    - For algorithm: pseudo-code, code or description similar to above gets full mark
    - Major logical errors → -4
    - Minor logical errors → -2
    - For algorithm; brute force approaches that fills from bottom to top end and try to handle the unfilled space → no marks.
    - Not mentioning the split (the divide step) and proceeding from 2 or more corners to fill in until empty space is left in center is actually brute force → no marks.

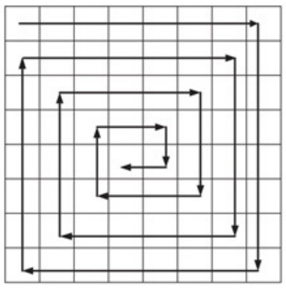
Part B: Greedy Algorithm Design

Assignment 1.4 [*25 marks*]

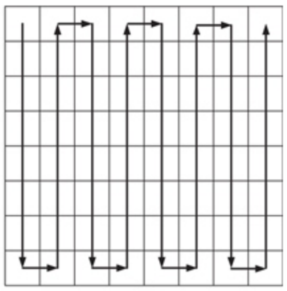
The tower in chess (tabyaa….) can move either horizontally to any square that is in the same row or vertically to any square that is in the same column as its current position. What is the minimum number of moves needed for the tower to pass over all the squares of an *n* x *n* chessboard? A tour does not have to start and end at the same squares, the squares where it starts, and ends are considered passed over by default.

Solution& marking

- *Observations:*
  - An optimal tour can start in the top left corner and following the greedy strategy to proceed as far as it can before making a turn. The resulting tour for the 8 x 8 board is shown below.



  - The number of moves in such a tour on the *n* x *n* board is 2*n*-1 which is the minimum for any *n*.
  - The above solution is not unique. One of the alternative solutions for the 8 x 8 board is shown below.



- *Marking*
    - Description similar to above with 2n-1 as an answer gets full mark
    - Major logical errors → -4
    - Minor logical errors → -2
    - While starting from center is allowed, it has less greedy merit and sides to being brute force. Since the goal is to have the most (greedy) coverage before changing directions and that is not applicable in this case as many turns have to be taken to continue to unexplored space → -6 marks.

Assignment 1.5 [*30 marks*]

There are *n* people, each in possession of a different part of a solution to a puzzle. They want to share the mini solutions they have with each other by sending electronic messages so that everyone can put them together to get the complete solution. What is the minimum number of messages they need to send to guarantee that everyone of them gets all the mini-solutions? Assume that the sender includes all the mini-solutions he or she already got and that a message may only have one addressee.

Solution& marking

- *Observations:*
    - A greedy approach seeks to increase as much as possible the total number of known solutions after each message sent.
    - The minimum number of messages is equal to $2n–2$ because an increase of the number of persons by one requires at least two extra message; to and from the extra person.
- *Solution:*
    - Configuration: Number the persons from 1 to *n*.
    - *Algorithm:* send the first *n* - 1 messages as follows: from 1 to 2, from 2 to 3, and so on, until the message combining the solutions initially known to persons 1, 2,..., *n*-1 is sent from person *n*. Then, send the message combining all the *n* solutions from person *n* to persons 1, 2, ...,*n*-1
- *Marking:*
    - Psudeo-code, code or description similar to above gets full mark
    - Major logical errors → -5
    - Minor logical errors → -2
    - There is another approach which is designate one person, say person 1, to whom everybody else sends the messages with the solution they know. After receiving all these messages, person 1 combines all the

solutions and sends them to the other *n*-1 persons. While this solution solves the problem, the greedy merit is debatable, but we won't consider it as wrong. Instead, we will leave it to student to justify the rationale.

- Absence of justification → -4.
- Unclear justification →-3.

Assignment 1.6 [*30 marks*]

You would like to help a veggie merchant by giving him a set of weights he can use to weight his products for the customers. Find an optimal set of n weights $\{w_1, w_2,\ldots,w_n\}$ so that it would be possible to weigh on a two-pan balance scale an integral load in the largest possible range from 1 to W, assuming the following:

(a) weights can be put only on the free pan of the scale.

Solution &marking

- *Observations:*
  - Let us apply the greedy approach to the first few instances of the problem;
    - For n=1, we have to use w_1 = 1, to balance weight 1.
    - For n=2, we add w_2 = 2, to balance the first previously unattainable weight of 2. The weights {1, 2} can balance every integral weight up to their sum 3.
    - For n=3, in the spirit of greedy thinking, we take the next previously unattainable weight w_3=4. The three weights {1,2,4} allow to weight any integral load l between1 and their sum 7, with l's binary expansion indicating the weights needed for l.

load l	1	2	3	4	5	6	7
l in binary	1	10	11	100	101	110	111
weights for load l	1	2	2 + 1	4	4 + 1	4 + 2	4 + 2 + 1

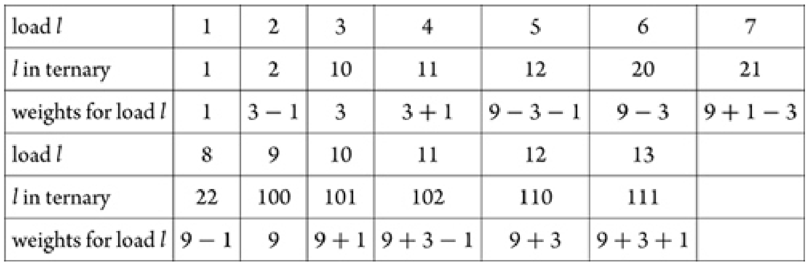
- *Solution:*
  - For any positive integer n the set of consecutive powers of 2 $\{w_i = 2^i, i=0,1,\ldots,n-1\}$ makes it possible to balance every integral load in the largest possible range from 1 to their sum which is equal to $2^n - 1$ inclusive.

        - The fact that every integral weight *l* in the range 1 <= *l* <= 2^n– 1 can be balanced with this set of weights follows immediately from the binary expansion of *l*, which yields the weights needed for weight *l*.
        - Since for any set of *n* weights, there are only 2^n - 1 different subsets (less if some of the weights are the same)that can be put on one pan of the scale, no more than 2^n - 1different loads can be measured with them. This proves that no set of *n* weights can cover a larger range of consecutive integral loads than 1 <= *l* <= 2^n - 1 if weights can be put only on the free pan of the scale.
- *Marking:*
    - Description, pseudo-code, code similar to above gets full mark
    - Mark this out of 12/30 since this is easier to solve
    - Major logical errors → -4
    - Minor logical errors → -2

(b) weights can be put on both pans of the scale.

Solution &marking

- *Observations:*
    - If weights can be put on both pans of the scale, a larger range can be reached with *n* weights for *n* > 1.
        - For *n* =1, the single weight still has to be 1.
        - The weights {1,3} enable weighting of every integral load up to 4.
        - The weights {1,3,9} enable weights of every integral load up to 13, as shown in the following table



load l	1	2	3	4	5	6	7
l in ternary	1	2	10	11	12	20	21
weights for load l	1	$3-1$	3	$3+1$	$9-3-1$	$9-3$	$9+1-3$
load l	8	9	10	11	12	13	
l in ternary	22	100	101	102	110	111	
weights for load l	$9-1$	9	$9+1$	$9+3-1$	$9+3$	$9+3+1$	

- If the ternary expansion of load *l*, $l <= (3^n - 1)/2$, contains only 0's and 1's, the load required putting the weights corresponding to the 1's on the opposite pan of the balance.
- If the ternary expansion of *l* contains one or more 2's, we can replace each 2 by (3-1) to represent *l* uniquely in the balanced ternary system.
  - For example;

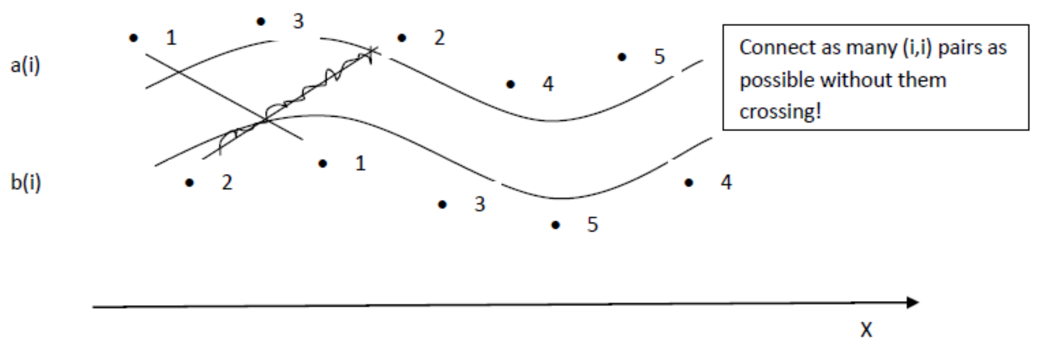
$$5 = 12_3 = 1 \,.\, 3^1 + 2 \,.\, 3^0 = 1 \,.\, 3^1 + (3 - 1) \,.\, 3^0 = 2 \,.\, 3^1 - 1 \,.\, 3^0$$
$$= (3 - 1) \,.\, 3^1 - 1 \,.\, 3^0 = 1 \,.\, 3^2 - 1 \,.\, 3^1 - 1 \,.\, 3^0$$

- *Solution*
  - The weights $\{w_i = 3^i, i=0,1,2,....,n-1\}$ enable weighting of every integral load from 1 to their sum inclusive which is $(3^n - 1)/2$.
- *Marking:*
  - Description, pseudo-code, code similar to above gets full mark
  - Mark this out of 18/30 since this is harder to solve
  - Major logical errors → -4
  - Minor logical errors → -2

Part C: Dynamic Programming Algorithm Design

Assignment 1.7 [*20 marks*]

Consider a 2-D map with a horizontal river passing through its center. There are *n* cities on the northern bank with x-coordinates a(1) ... a(*n*) and n cities on the southern bank with x-coordinates b(1) ... b(*n*). You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city *i* on the northern bank to city *i* on the southern bank. Using DP problems studied in class (if you wish), give a dynamic programming solution to solve this problem.


a(i)
b(i)
1
3
2
5
4
1
2
3
5
4
Connect as many (i,i) pairs as possible without them crossing!
X

Solution

- Run Longest common subsequence on a() and b()

Assignment 1.8 [*25 marks*]

Consider the case of 3 knapsacks, all of the same maximum weight w, where w is greater than the sum of some 1/3 of the items given. Your task is to divide a given set of items into 3 sets of equal value to be able to put each set in one of the bags. You can assume that the following invariant holds on the items:

The input items can be split into 3 subsets of equal value and equal weight.

Design a dynamic programming solution which identifies the subsets.

Solution& marking

- *Observations:*
    - This problem is similar to the classical 1/0 knapsack with 2 modifications;
        - An item is picked once, so we need to determine for any particular item whether it was previously included as part of the solution to an earlier knapsack or not
        - actual weight of each knapsack is going to be 1/3 of the total weight of all the items – instead of considering it as a bigger value.
- *Marking:*
    - Those who solved correctly optimizing both (could be done using 3D array) will be given a bonus of 15 marks.

Assignment 1.9 [*30marks*]

John, a tourist currently in Cairo, decided to go on a road trip from Cairo to Aswan! He has a saving of 2,000EGP and is ready to spend it all. There are *n* poll stations connecting Cairo to Aswan along the river side, referred to as r_1, r_2, …,r_n. There are also *n* poll stations connecting Cairo to Aswan through oasis in the desert, referred to as o_1, o_2,…, o_n. To reach his destination, he has to pass by *n* poll stations, where poll station *i* is either is either r_i or o_i. In other words, John has to pass by *n* poll stations, where poll station*i* ($1 <= i <= n$) can be located either on the river side or beside an oasis.

The total amount of money spent in his journey will be the cost of the fuel from one poll station to the next in addition to the taxes paid at every poll station he passes by (which differs from one poll station to the other). John wants to minimize the cost of his journey. He drew a map of all the possible routes between poll station *i* and poll station *j* where $j = i + 1$, and calculated the fuel cost for those routes.

He also inquired about the amount of taxes paid at each poll station. John wants to calculate the cost of the cheapest route from Cairo to Aswan, to know whether he can afford the trip or not.

Your are given:

riverToRiver(*i*,*j*): the cost of fuel needed to go from poll station r_i to poll station r_j .

oasisToOasis(*i*,*j*): the cost of fuel needed to go from poll station o_i to poll station o_j .

riverToOasis(*i*,*j*): the cost of fuel needed to go from poll station r_i to poll station o_j .

oasisToRiver(*i*,*j*): the cost of fuel needed to go from poll station o_i to poll station r_j .

riverTaxes(*i*): the amount of taxes paid at poll station r_i.

oasisTaxes(*i*): the amount of taxes paid at poll station oi.

Using dynamic programming, develop a solution that decides whether John the tourist can hit the road, or should stay at hotel.

Solution&marking

- *Recursive Formulation:*

$$\text{River[i]} = \begin{cases} riverTaxes(i) & \text{if } i = 1; \\ min\begin{cases} \text{River[i-1]} + \text{riverToRiver(i-1,i)} + \text{riverTaxes(i)} \\ \text{Oasis[i-1]} + \text{oasisToRiver(i-1,i)} + \text{riverTaxes(i)} \end{cases} & \text{if } 2 \leq i \leq n \end{cases}$$

$$\text{Oasis[i]} = \begin{cases} oasisTaxes(i) & \text{if } i = 1; \\ min\begin{cases} \text{River[i-1]} + \text{riverToOasis(i-1,i)} + \text{oasisTaxes(i)} \\ \text{Oasis[i-1]} + \text{oasisToOasis(i-1,i)} + \text{oasisTaxes(i)} \end{cases} & \text{if } 2 \leq i \leq n \end{cases}$$

- *Pseudo-Code:*

```
function GETCOST(int n)
    River ← (int) AllocateArray(n)
    Oasis ← (int) AllocateArray(n)
    $i \leftarrow 1$
    while $i \leq n$ do
        if $i = 1$ then
            $River[1] \leftarrow riverTaxes(1)$
            $Oasis[1] \leftarrow oasisTaxes(1)$
        else
            $River[i] \leftarrow min(River[i-1] + riverToRiver(i-1, i) + riverTaxes(i),$
                                  $Oasis[i-1] + oasisToRiver(i-1, i) + riverTaxes(i))$
            $Oasis[i] \leftarrow min(River[i-1] + riverToOasis(i-1, i) + oasisTaxes(i),$
                                  $Oasis[i-1] + oasisToOasis(i-1, i) + oasisTaxes(i))$
        end if
        $i \leftarrow i + 1$
    end while
    if $River[n] < 2,000$ or $Oasis[n] < 2,000$ then
        PRINT("Hit the road!")
    else
        PRINT("Stay at home!")
    end if
end function
```

- *Marking:*
    - Major logical errors → -5
    - Minor logical errors → -2

Assignment 1.10 [*35 marks*]

Design a dynamic programming algorithm to count how many different ways the palindrome

Was it a cat i saw

can be read in the diamond-shaped arrangement shown below.

```
            W
          W A W
        W A S A W
      W A S I S A W
    W A S I T I S A W
  W A S I T A T I S A W
W A S I T A C A T I S A W
  W A S I T A T I S A W
    W A S I T I S A W
      W A S I S A W
        W A S A W
          W A W
            W
```

You may start at W and go in any direction on each step, up, down, left, right through adjacent letters. The same letter can be used more than once in the same sequence.

Solution

- *Observations*
  - We can start by counting the number of ways to spell CAT I SAW. Any such string starts at the center and is contained in one of the four triangles formed by the diamond's diagonals. One of these triangles is shown below.

```
            W
          W A W
        W A S A W
      W A S I S A W
    W A S I T I S A W
  W A S I T A T I S A W
W A S I T A C A T I S A W
  W A S I T A T I S A W
    W A S I T I S A W
      W A S I S A W
        W A S A W
          W A W
            W
```

W_1

$A_1 W_6$

$S_1 A_5 W_{15}$

$I_1 S_4 A_{10} W_{20}$

$T_1 I_3 S_6 A_{10} W_{15}$

$A_1 T_2 I_3 S_4 A_5 W_6$

$C_1 A_1 T_1 I_1 S_1 A_1 W_1$

- The number of strings spelling CAT I SAW can be computed by diagonals parallel to the triangle's hypotenuse by adding up the adjacent numbers to the left and below the letter in question. These numbers from Pascal's triangle. The sums of these numbers on the triangle's hypotenuse - the diamond's border - is 2^6.
- The total number of strings spelling CAT I SAW for the entire diamond is then obtained by the formula $4 \times 2^6 - 4$ (we need to subtract 4 to compensate the over count of the strings along the diamond's diagonals). This implies that the total number of ways to spell WAS IT A CAT I SAW is given by the formula $(4 \times 2^6 - 4)^2 = 63{,}504$
- *Marking:*
    - Description, pseudo-code, code similar to above gets full mark
    - Justification is important $\rightarrow$ if none provided and just the number $\rightarrow$ *no marks*
    - Mistakes in counting due to missing cases $\rightarrow$ -10 to -6 depending on how much is missed.
    - Other major logical errors $\rightarrow$ -5
    - Minor logical errors $\rightarrow$ -3

Part D: Open Problems

The decision is yours regarding which algorithm design technique should be used to solve each of the below problems.

Assignment 1.11 [*30 marks*]

A firm has invented a super-strong glass sheet. For publicity purposes, it wants to determine the highest floor in a 100-story building from which such a glass can fall without breaking. The firm has given a tester two identical glass sheets to experiment with. of course, the same glass sheet can be dropped multiple lines unless it breaks. What is the minimum number of droppings that is guaranteed to determine the highest safe floor in all cases?

Solution

- *Observations*
    - This is a greedy problem because we want to go to highest floor possible.
    - Let H(k) be the maximum number of floors for which the problem can be solved in k drops.
    - The first drop has to be made from floor k because if the glass breaks, each of the lower k-1 floors will be needed to test sequentially starting with the first floor.
    - If the first drop does not break the glass, the second drop has to be made from floor k + (k-1) to be prepared for the possibility that if the glass breaks, each of the k -2 floors from floor k + 1 to floor 2k -2 need to be tested sequentially.
    - On repeating this argument for the remaining k - 2 drops, we obtain the following formula for H(k)

$$H(k) = k + (k - 1) + ... + 1 = k\,(k+1)/2$$

- What remains to be done to answer the question is to find the smallest value of *k* such that *k* (*k*+1) /2 >= 100. This value is *k* = 14.

- *Algorithm*
  - The first glass sheet can be dropped from floors 14,27,39,50,60,77,84,90,95 and 99 and 100 until it breaks, and if this happens, the second sheet is to be dropped from each consecutive floor starting with the next floor after the last successfully tested one.
    - Note that this solution is not unique; the first drop can also be executed from floors 13, 12, 11, and 10, etc…
- *Marking:*
  - Description, pseudo-code, code similar to above gets full mark
  - Major logical error → -5
  - Minor logical error → -3

Assignment 1.12 [*30 marks*]

Given a list of *n* integers, $v_1 \ldots v_n$, the product-sum is the largest sum that can be formed by multiplying adjacent elements in the list. Each element can be matched with at most one of its neighbors. For example, given the list 1, 2, 3, 1 the product sum is 8 = 1 + (2 × 3) + 1, and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is 19 = (2 × 2) + 1 + (3 × 2) + 1 + (2 × 2) + 1 + 2. Develop an algorithm to find the largest product sum [the 8 or the 19 in examples].

Solution

- *Observations*
  - Dynamic programming is the correct design choice since it will yield fastest performance.
  - One-D array is enough to compute the answer
- *Algorithm*
  - *Recurrence*

$$OPT[j] = \begin{cases} \max\{OPT[j-1] + v_j, OPT[j-2] + v_j * v_{j-1}\} & \text{if } j \geq 2 \\ v_1 & \text{if } j = 1 \\ 0 & \text{if } j = 0 \end{cases}$$

  - *Pseudo-code*

```
$Prod\text{-}Sum(\textbf{int}[\,]v, n)$
  if $(n == 0)$ return $0$;
  int [ ] $OPT$ = new int $[n+1]$;
  $OPT[0] = 0$;
  $OPT[1] = v[1]$;
  for int $j = 2$ to $n$
    $OPT[j] = \max(OPT[j-1] + v[j], OPT[j-2] + v[j] * v[j-1])$;
  return $OPT[n]$;
```

Assignment 1.13 [*40 marks*]

Consider nested function calls, such as f(g(x), h(y, k())), and how they are evaluated by an interpreter. Suppose that functions require one time step for each nested function call they make, and only one function call can be made at each time step. Then the example expression above can be evaluated in four steps, as follows:

- time 0: call f() - now we know that we have to call g() and h(),
- time 1: f calls g(),
- time 2: f calls h() - now we know that we have to call k(),
- time 3: h calls k(),
- time 4: all done!

You should have no trouble convincing yourself that every order of calls will require the same number ofsteps (for example, if f calls h() first, it does not change the other calls that must be made, just their order).

Now suppose that we have access to a parallel computer that can execute multiple functions calls at the same time. It is still the case that functions require one time step for each nested function call they make but now different functions can make calls at the same time. Then, the example above can be evaluated in only
three steps - we cannot do better, because f requires two time steps (one to call h and one to call g):

- time 0: call f() - now we know that we have to call g() and h(),
- time 1: f calls h() - now we know that we have to call k(),
- time 2: f calls g() and h calls k(), in parallel,
- time 3: all done!

This situation can be represented by the following abstract "Efficient Function Calls" problem.

Input: A nested function call expression E of the form "f(e_1,….e_k)," where f is a function name and each e_i is either a variable or another nested function call expression (it is possible to have *k* = 0).

Output: The order of nested calls for each function in E, to minimize the total number of steps required to evaluate the expression on the parallel computer.

Give an algorithm to solve this problem efficiently: include a brief high-level English description, as well as a detailed pseudo-code implementation. Justify that your algorithm is correct, and analyze its worst-case running time.

Solution& marking

- *Observations:*
  - This is an optimization problem, hence dynamic programming is the correct choice.
  - Let us try to describe the recursive structure of the sub-problems;
    - Consider an input $E = f(e_1, e_2, \dots, e_n)$.
    - Note that if any e_i is a variable, then it has no effect on the evaluation time. So without loss of generalization,

      let $e_i = g_i(\dots)$ for each i.
    - Any optimal solution for this input has the following form, where $i_1, i_2, \dots, i_n$ is a permutation of $[1, 2, \dots, n]$:

      time 0: call $f()$

      time 1: f calls $g_{i1}()$

      time 2: f calls $g_{i2}()$, . . .

      . . .

      time n: f calls $g_{in}()$, . . .

      . . .
    - The overall evaluation time for this solution is

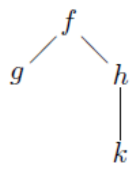
      $$T(f) = \max\{\, 1 + T(g_{i1}), 2 + T(g_{i2}), \dots, n + T(g_{in}) \,\}$$

    where $T(g_{ij})$ is the evaluation time for input $g_{ij}(\dots)$ [i.e. performing the steps in the call to g_{ij} in the same order as in the overall solution].

  - In this solution, it is possible that some of the calls to functions g_i are performed in a sub-optimal manner because they may not contribute to raise the overall maximum. But it not possible for every call to g_i to be done in a sub-optimal manner, otherwise, we could improve on every one of those calls and lower the overall completion time. This means that an optimal solution to the original problem can always be obtained from optimal solutions to the sub-problems $g_1, g_2, \ldots, g_n$.
  - Hence, ordering the calls to the g_i's so that

    $T(g_{i1}) >= T(g_{i2}) >= \ldots >= T(g_{in})$ minimizes $T(f)$.

- *Algorithm:*
  - Define an array that stores optimal values for arbitrary sub-problems. There is no simple linear structure to specify sub-problems. However, every function call expression E can be represented as a tree, with one node for each function symbol in E, where the children of node f are exactly the functions called directly by f variables are ignored. For example, the expression $f(g(x), h(y,k()))$ is represented by the tree below. For each function symbol f in the expression E, define $T[f]$ to be the minimum evaluation time for the sub-expression rooted at f.



  - Defining a recurrence relation for the array values.
    - For every leaf f in the tree representation for E,

      $T[f] = 1$.

    - For every internal node $f(g_1, \ldots, g_n)$ in the tree,

      $T[f] = \max\{1+T[g_{i1}], \ldots, n+T[g_{in}]\}$

      where $i_1, \ldots, i_n$ is a permutation of $[1, \ldots, n]$ such that

$T[g_{i1}] >= \ldots >= T[g_{in}]$.

- Writing a bottom-up algorithm to compute the array values, following the recurrence.
  - Construct the tree representation for *E* -- call it *R*
  - Perform a post-order traversal of R -- for each node *f*:

```
if $f$ is a leaf:
        $T[f] = 1$
else:
        let $g_1, g_2, \ldots, g_k$ be the children of $f$, ordered so that
                $T[g_1] > T[g_2] > \ldots > T[g_k]$
        $T[f] = \max\{1+T[g_1], 2+T[g_2], \ldots, n+T[g_k]\}$
```

- Use the computed values to reconstruct an optimal solution. Starting at the root of *R*, for each *f* in *R* with children $g_1, g_2, \ldots, g_k$, call the g_i's in non-increasing order of $T[g_i]$.

- *Marking:*
  - Description, pseudo-code, or code similar to above gets full mark
  - Pure divide and conquer approaches will not always yield optimal solution → -10 marks.
  - Hybrid divide and conquer approaches, aka guided by heuristics, should be inspected for optimality.
  - Major logical error → -5
  - Minor logical errors → -3