CSC373 Assignment 1

Note: all pseudo-code will be written using Python-style semantics

1. Finding the "edge" of an infinite array (renaming *A* to *unbounded_arr*)
    a. English Description
        i. Keep doubling our current index (starting from 1) until we hit the first infinity – when we stop we will have a bounded array
        ii. Check the middle two elements of the current array section
            1. If they are of form [int, int], repeat step ii on the right half of the array
            2. If they are of form [infinity, infinity], repeat step ii on the left half of the array
            3. If they are of form [int, infinity], return the index that we checked
    b. Pseudocode

```
# Let f ~ g denote whether or not arrays f, g have the same form
# Example: [1, 2] ~ [int, int] → True
# Example: [1, pi] ~ [int, real] → True
# Example: [e, pi] ~ [int, int] → False

def find_edge(unbounded_arr):

	# Where we currently are in the array
	cur_ind = 1

	# Hop up to the first position that contains infinity
	while unbounded_arr[cur_ind] != infinity:
		cur_ind *= 2

	# This is the array we know the edge to be inside
	bounded = unbounded_arr[:cur_ind]

	# Midpoint of the array we are looking at
	mid_ind = len(bounded) // 2
	mid_pair = bounded[mid_ind:(mid_ind + 1)]

	# Look for where the edge occurs
	# (note that infinity section is right of the int section)
	while not (mid_pair ~ [int, infinity]):

		# We're looking in the infinity-only section
		if mid_pair ~ [infinity, infinity]:

			# Look in the left half of the array
			mid_ind = mid_ind // 2
			mid_pair = bounded[mid_ind:mid_ind + 1]

		# We're looking in the int-only section
		elif mid_pair ~ [int, int]:

			# Look in the right half of the array
			mid_ind = mid_ind + (mid_ind // 2)
			mid_pair = bounded[mid_ind:mid_ind + 1]

	return mid_ind
```

c. Useful loop invariants
    i. First loop
        1. Loop invariant
            a. After the $j-th$ iteration of this loop, *cur_ind* $= 2^{j-1} < n$.
        2. Termination condition
            a. When this loop terminates on some j, we know that $arr[2^j] = \infty$ (by the continuation condition of the loop)
        3. Implications for correctness
            a. By the conditions laid out in the question, we also know that any index greater than n has value ∞, which means that $2^j > n$ (when the algorithm terminates after iteration j).
            b. Therefore, we know that the upper-bound on the array is 2^j, which is at most $2n - 2 \in O(n)$ (in the worst case where the second-last *cur_ind* is $n - 1$, just before n and we double cur_ind)
            c. Therefore, we know that the upper bound on the array will produce a bounded array of size $O(n)$ that definitely contains n.
    ii. Second loop
        1. Loop invariant
            a. Let r be the length of *bounded_arr*
            b. After the $j-$ th iteration of this loop, we are currently observing a section of $bounded\_arr$ that is of length $\frac{r}{2^j}$, and this section of $bounded\_arr$ is centered around or *mid_pair*
        2. Termination condition
            a. This loop will eventually terminate when *mid_pair* is of the form [int, infinity]
        3. Implications for correctness
            a. Since we know that:
                i. *n* is in *bounded_arr*
                ii. [bounded_arr[n], bounded_arr[n + 1]] will be the only pair that has form [int, infinity]
            b. We know that the algorithm will terminate when it "hits" n and returns it
d. Runtime Analysis
    i. The algorithm only makes $O(\log n)$ array accesses in the first part
        1. The algorithm will "hop up" to at most $2n - 2$
            a. In the worst case, the second-last index to be checked in this half will be directly left of the edge (i.e. index $n - 1$)
            b. In this case, the algorithm will double the second-last index to $2n - 2$, and then stop as the value at the current index is infinity
                $$(2n - 2 > n \Rightarrow arr[2n - 2] = \infty)$$
        2. We note that the number of times 1 has to be doubled to reach k is some i such that:
            $$1 * 2^i = k$$
            $$\Rightarrow 2^i = k$$
            $$\Rightarrow i = \log_2 k$$
            Thus the number of times we can double 1 to reach k is $\log k$. Thus:
            $$k = 2n - 2$$
            $$\Rightarrow i = \log_2(2n - 2)$$

$$\Rightarrow i \in O(\log n)$$

3. Thus, the total number of while-loop iterations that can be made is $O(\log n)$
4. Since one access is made per "hop" (in the while-loop continuation check), the number of array accesses must be equal to the number of while-loop iterations
5. Thus, the number of array accesses must be $O(\log n)$ to find the upper bound on the array in the first part of the algorithm

ii. The algorithm only makes $O(\log n)$ array accesses in the second part

1. In the worst case, the algorithm will have to keep halving *bounded* until it only has to observe a list of length 2, in which case it has to terminate, as this list must display the edge of the array (because we know the edge is inside *bounded,* in the worst case it has to be the final pair (provided we are looking in the proper halves)).
2. We note that i, the number of times k has to be halved to get down to 2 is:

$$k * \left(\frac{1}{2}\right)^{i} = 2$$
$$\Rightarrow \log_2\left(k * \left(\frac{1}{2}\right)^{\mathrm{i}}\right) = \log_2 2$$
$$\Rightarrow \log_2 k + \log_2\left(\frac{1}{2}\right)^{i} = 1$$
$$\Rightarrow -i = 1 - \log_2 k$$
$$\Rightarrow \mathrm{i} = \log_2 \mathrm{k} - 1$$
$$\Rightarrow i \in O(\log k)$$

3. By our reasoning for the first-half of the algorithm, *bounded* must be of size $O(\mathrm{n})$ in any case (since maximum possible size is $2n - 2$)
4. Thus, this worst-case halving will take $O(\log \mathrm{n})$ while-loop iterations, and since every iteration of the while-loop makes two array accesses (one for each element in the middle pair), the number of accesses must be $O(2 \log \mathrm{n}) \in O(\log \mathrm{n})$

iii. If the runtime of both parts is $O(\log n)$, the algorithm's total runtime $T(n)$ will be as follows:

$$T(n) = \text{Runtime of Part 1} + \text{Runtime of Part 2}$$
$$\Rightarrow T(\mathrm{n}) \in O(\log n) + O(\log \mathrm{n})$$
$$\Rightarrow T(n) \in O(\log n)$$

2. Detection of $\frac{n}{2}$ occurrences (majority element) in an array (renaming *A* to *arr*)
    a. English Description
        i. Base case of the algorithm is empty array: return the tie-breaker (default value is None)
        ii. Go through the array pairwise*, add elements from matching pairs to a holding array, and if there is a left-over element, it is a tie-breaker
        iii. Repeat this procedure on the holding array to yield a candidate value and then count the number of candidate occurrences in the original array
        iv. If the number of candidate occurrences is greater than half the array length, we have clearly found the majority (use the tie-breaker to break a tie where count is exactly half)
        v. Return whether or not this recursive procedure yielded a value that is not None

    *Pairwise: one way of going through $[A, B, C, D]$ pairwise would be $(A, B), (C, D)$ – we cannot consider an element twice (e.g. once (A, B) is considered, we cannot consider any other pairs containing A or B)

b. Pseudocode

```
# The answer to the question
def exists_majority(arr):
       return (majority_element(arr, None) is not None)

# Finds the majority element (frequency > n/2) in the array
def majority_element(arr, tie_breaker):

    # Base case, it has to be the tie-breaker
    if len(arr) == 0:
        return tie_breaker

    # Otherwise, we have to do the pair elimination strategy
    else:

        # If the array is of odd length, there is going to be a tie-breaker
        if len(arr) % 2 == 1:
            tie_breaker = arr[-1]

        # Eliminate non-matching pairs
        holder = []
        for i in range(0, len(arr) - 1, 2):
            if arr[i] == arr[i + 1]:
                holder += [arr[i]]

        # Look for the potential majority element in the reduced pairs
        maj_candidate = majority_element(holder, tie_breaker)

        # We couldn't find anything that occurred more than n/2 times
        if maj_candidate is None:
            return None

        # We found something that may have occurred more than n/2 times
        else:
            # Count the number of times the candidate occurs
            freq_pot_maj = count(arr, maj_candidate)

            # Trivially has to be a majority
            if (is_majority(arr, maj_candidate)):
                return maj_candidate

            # Otherwise if the candidate:
            #   Takes up 50% of the even-length array
            #   Is equal to the tie breaker
            # The tie is broken (50% + 1 > 50%)!
            elif (freq_pot_maj == len(arr) / 2 and maj_candidate == tie_breaker):
                return maj_candidate

            # Otherwise, we have an element that is not the majority (count < n/2)
            else:
                return None

# Counts the number of occurences
def count(arr, target):
    ct = 0
    for num in arr:
        if num == target:
            ct += 1
    return ct

# Checks if target is majority element
def is_majority(arr, target):
    ct = count(arr, target)
    return (ct > (len(arr) / 2))
```

c. Correctness Argument
    i. If x is the majority value, it must occur more than $n/2$ times in *arr*
    ii. Thus, we can make the following conclusions about eliminating dissimilar pairs of values:
        1. If we eliminate a pair that contains x, then the other element would have to be some other value y. Since only x can be the majority, we have preserved majority by removing an x and a non-x value.
        2. If we eliminate a pair does not contain x, we maintain majority because the contribution of non-x values is irrelevant to the majority property (i.e. decreasing the denominator of a fraction increases the overall value of the fraction)
        3. If we keep a pair, it means that both values were x, which means that we are preserving the majority element
    iii. Thus, if we know *holder*'s majority, we can make some inferences about *arr*'s majority by counting the number of times *holder*'s majority occurs in *arr*
        1. If the number of occurrences is less than the majority threshold $\left(\frac{n}{2}\right)$, there is no majority element
        2. If the number of occurrences is equal to the majority threshold, we must decide using the tie-breaker variable
        3. If the number of occurrences is greater than the majority threshold, we clearly have a majority value

d. Recurrence

Note: I can't find a proper piecewise equation tool in MS word

The recurrence relation for this algorithm is:

$$(n = 0) \Rightarrow \mathrm{T(n)} = 0$$
$$(n > 0) \Rightarrow T(n) = T\left(\frac{n}{2}\right) + O(n)$$

Using master theorem, this means (for recurrence relations of form $T(\mathrm{n}) = aT\left(\frac{n}{b}\right) + O(n^{\mathrm{d}})$):

$$(a, b, d) = (1, 2, 1)$$
$$\Rightarrow \mathrm{a\ vs.\ b^d} \Rightarrow 1 \text{ vs } 2^1 \Rightarrow 1 < 2 \Rightarrow \mathrm{a < b^d}$$
$$\Rightarrow \left(\left(T(n) \in \Theta\left(\mathrm{n^d}\right)\right) \wedge (\mathrm{d} = 1)\right)$$
$$\Rightarrow \mathrm{T(n)} \in \Theta(\mathrm{n})$$

3. Profit-maximizing job scheduling
    a. English Description
        i. Sort the jobs in order of decreasing profit (e.g. highest profit first)
        ii. Go through this profit-sorted list of jobs and schedule each job as late as possible
    b. Pseudocode

```
# The answer to the question
def maximize_profit(D, P):

        # Merge the jobs into an array of tuples so it's easier to sort
        tups = []
        for i in range(len(D)):
                tups += [(D[i], P[i])]

        # Sort the job tuples by profit, decreasing (second element of each tuple)
        tups = sorted(tups, key = lambda x: x[1], order = decreasing)

        # This is the schedule we are constructing
        sched = []

        # Add the most profitable job as late as it can possibly go
        for job in tups:
                add job to sched such that the following properties are satisifed:
                        1)  Job is scheduled as late as possible
                        2)  Job is scheduled at least 1 hour before its deadline
                        3)  Job does not conflict with other jobs, i.e. its start time cannot
                            come before any other job's endtimes

        # Return the schedule that we made
        return sched
```

    c. Formal Correctness (by induction)

        I do not know how to answer this question

    d. Running Time Analysis
        i. This algorithm will take at least $O(n^2)$ time. This is because the algorithm has two main parts:
            1. Sorting by profit – this will take $O(n\log n)$ time with optimal sorting algorithms (e.g. merge-sort)
            2. Picking jobs such that they don't conflict – this will take $O(n^2)$ time because in the worst case, each job will have to be compared against $O(n)$ other jobs to find an appropriate start time. Since there are $O(n)$ jobs being compared against $O(n)$ jobs, this will take $O(n^2)$ time
            3. Thus, the total runtime should be: $O(nlogn) + O(\mathrm{n}^2)$, but since
            $$O(nlogn) \in O(n^2) \Rightarrow \mathrm{O(nlogn)} + O(\mathrm{n}^2) \in O(n^2)$$
            4. Therefore, the total running time will be $O(n^2)$

4. Efficient triangulation

Let $i, j, k \in \{1, 2, \ldots n\}$.

Let $d(p_1, p_2) = \left(\left(\mathrm{p}_{1_\mathrm{x}} - \mathrm{p}_{2_\mathrm{x}}\right)^2 + \left(\mathrm{p}_{1_\mathrm{y}} - \mathrm{p}_{2_\mathrm{y}}\right)^2\right)^{\frac{1}{2}}$ (distance from p_1 to p_2)

Let $T(\mathrm{P})$ denote the triangulation of polygon P

a. Semantic Array
   i. $C[i, j] =$ the efficient length of triangulating a polygon composed of points A[i] **through** A[j] (these are consecutive points)
   ii. Answer: contained in $C[1, n]$
b. Computational Array
   i. $C[i, j] = 0$ if $\mathrm{k} < \mathrm{i} + 2$ – base case
   ii. $C[i, j] = \min_{\mathrm{k} \in (\mathrm{i,j})} \{\mathrm{C[i,k]} + \mathrm{C[k,j]} + \mathrm{d(A[i], A[k])} + \mathrm{d(A[j], A[k])}\}$
c. Array Equivalence
   i. Base case: there's no way to triangulate 2 or less points, so 0 is what we return
   ii. Inductive case
      1. Note that we can split a polygon P composed of points $p_i, \ldots p_k, \ldots, p_j$ into two separate polygons P_L, P_R by splitting the points on some index k
      2. Thus we let:
         a. $P_\mathrm{L} =$ polygon composed of points $p_i, \ldots p_k$
         b. $P_R =$ polygon composed of points $p_k, \ldots, p_j$
      3. Note that we can merge two triangulations into one triangulation by drawing two lines from one point on one triangulation to two points on another triangulation, thereby creating another triangle that connects the two triangulations
      4. To find the most efficient triangulation of polygon $P, T_{eff}(\mathrm{P})$, we need to do the following:
         a. Figure out the efficient triangulations for all possible P_L, P_R
         b. Draw the connecting triangles between $T_{eff}(\mathrm{P_L})$ and $T_{eff}(\mathrm{P_R})$ by connecting point k to points i and points j
         c. Figure out the resulting cost of the above drawing by adding the costs of $T_{eff}(\mathrm{P_L})$ and $T_{eff}(\mathrm{P_R})$, with the distance of the lines drawn between i, j, k.
         d. Find the lowest overall total cost over all possible P_L, P_R
         e. Note that the previous two steps are the English representation of $C[i, j]$, provided $k \geq i + 2$ (i.e. not base case)
d. Running Time Analysis
   i. Note the following ranges – the algorithm must do essentially 3 nested loops for each of the variables:
      1. $i \in [1, n]$
      2. $j \in [i + 1, n]$, but since i's minimum value is 1, $\Rightarrow j \in [2, n]$
      3. $k \in [i + 1, j - 1]$, but since i's minimum value is 1 and j's maximum value is n $\Rightarrow k \in [2, \mathrm{n} - 1]$
   ii. Since all of these ranges are size $O(n)$, their product must be $O(n) * O(n) * O(n) \Rightarrow O(\mathrm{n}^3)$. Thus, the algorithm runs in cubic time.

5. Profit-maximizing chocolate slicing

   Define $P(r, c)$ to be the profit gained from a chocolate bar with r rows and c columns (no breakage!) – assume that this is already pre-calculated and arranged in a 2D array for $O(1)$ access time

   a. Semantic Array
      i. $C[i, j]$ is the maximum profit able to be extracted from a chocolate bar with i rows and j columns
      ii. Answer: $C[W, H]$
   b. Computational Array
      i. $C[0, j] = 0, j \in (0, \mathrm{H})$ (base case 1)
      ii. $C[i, 0] = 0, \mathrm{i} \in (0, \mathrm{W})$ (base case 2)
      iii. $C[i, j] = \max_{\substack{w \in (1, W-1), \\ \mathrm{h} \in (1, \mathrm{H}-1)}} \left\{ \begin{matrix} \mathrm{P(i, j)} \\ \mathrm{C[i - w, j]} + \mathrm{C[w, j]} \\ \mathrm{C[i, j - h]} + \mathrm{C[i, h]} \end{matrix} \right\}$
   c. Array Equivalence
      i. There are three ways to split a chocolate bar:
         1. Don't split it at all
         2. Split it on the $w -$ th row
         3. Split it on the $h -$ th column
      ii. For these three cases, we could receive the following respective profits:
         1. $P(i, j)$, where our chocolate bar has i rows and j columns
         2. The maximum amount of profit we can extract out of the piece that goes to the $w -$ th row PLUS the maximum amount of profit we can extract out of the piece that goes beyond the $w - th$ row
         3. The maximum amount of profit we can extract out of the piece that goes to the $h -$ th column PLUS the maximum amount of profit we can extract out of the piece that goes beyond the $h - th$ column
      iii. In order to maximize our profit, we have to take the largest of these three values
      iv. Note that the previous two items essentially describe the computational array in plain English
   d. Running Time Analysis
      i. We note that the overall number of sub-problems is HW (area of the computational matrix). At each sub-problem, we have to consider W possible row breaks and H possible column breaks, plus a look-up for the profit of the intact bar. Thus, if the number of sub-problems is $O(HW)$ and the number of choices per sub problem is $O(HW) + O(1) \in O(\mathrm{HW})$, our total runtime is $O(HW) * O(HW) \Rightarrow O(H^2W^2)$.