**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*)
   1. English Description
      1. Keep doubling our current index (starting from 1) until we hit the first infinity – when we stop we will have a bounded array
      2. 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
   2. 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

* 1. Useful loop invariants
     1. First loop
        1. Loop invariant
           1. After the iteration of this loop, cur\_ind .
        2. Termination condition
           1. When this loop terminates on some , we know that (by the continuation condition of the loop)
        3. Implications for correctness
           1. By the conditions laid out in the question, we also know that any index greater than has value , which means that (when the algorithm terminates after iteration ).
           2. Therefore, we know that the upper-bound on the array is , which is at most (in the worst case where the second-last cur\_ind is , just before and we double )
           3. Therefore, we know that the upper bound on the array will produce a bounded array of size that definitely contains .
     2. Second loop
        1. Loop invariant
           1. Let be the length of bounded\_arr
           2. After the iteration of this loop, we are currently observing a section of that is of length , and this section of is centered around or mid\_pair
        2. Termination condition
           1. This loop will eventually terminate when mid\_pair is of the form [int, infinity]
        3. Implications for correctness
           1. Since we know that:

n is in bounded\_arr

[bounded\_arr[n], bounded\_arr[n + 1]] will be the only pair that has form [int, infinity]

* + - * 1. We know that the algorithm will terminate when it “hits” and returns it
  1. Runtime Analysis
     1. The algorithm only makes array accesses in the first part
        1. The algorithm will “hop up” to at most
           1. In the worst case, the second-last index to be checked in this half will be directly left of the edge (i.e. index )
           2. In this case, the algorithm will double the second-last index to , and then stop as the value at the current index is infinity

* + - 1. We note that the number of times has to be doubled to reach is some such that:

Thus the number of times we can double 1 to reach is . Thus:

* + - 1. Thus, the total number of while-loop iterations that can be made is
      2. 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
      3. Thus, the number of array accesses must be to find the upper bound on the array in the first part of the algorithm
    1. The algorithm only makes 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 , 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 , the number of times has to be halved to get down to is:
       3. By our reasoning for the first-half of the algorithm, bounded must be of size in any case (since maximum possible size is )
       4. Thus, this worst-case halving will take 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
    2. If the runtime of both parts is , the algorithm’s total runtime will be as follows:

1. Detection of occurrences (majority element) in an array (renaming A to arr)
   1. English Description
      1. Base case of the algorithm is empty array: return the tie-breaker (default value is None)
      2. 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
      3. Repeat this procedure on the holding array to yield a candidate value and then count the number of candidate occurrences in the original array
      4. 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)
      5. Return whether or not this recursive procedure yielded a value that is not None

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

* 1. 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))

* 1. Correctness Argument
     1. If is the majority value, it must occur more than times in arr
     2. Thus, we can make the following conclusions about eliminating dissimilar pairs of values:
        1. If we eliminate a pair that contains , then the other element would have to be some other value . Since only can be the majority, we have preserved majority by removing an and a non- value.
        2. If we eliminate a pair does not contain , we maintain majority because the contribution of non- 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 , which means that we are preserving the majority element
     3. 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 , 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
  2. Recurrence

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

The recurrence relation for this algorithm is:

Using master theorem, this means (for recurrence relations of form :

1. Profit-maximizing job scheduling
   1. English Description
      1. Sort the jobs in order of decreasing profit (e.g. highest profit first)
      2. Go through this profit-sorted list of jobs and schedule each job as late as possible
   2. 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

* 1. Formal Correctness (by induction)

I do not know how to answer this question

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

1. Efficient triangulation

Let .

Let (distance from to )

Let denote the triangulation of polygon

* 1. Semantic Array
     1. the efficient length of triangulating a polygon composed of points A[i] through A[j] (these are consecutive points)
     2. Answer: contained in
  2. Computational Array
     1. – base case
  3. Array Equivalence
     1. Base case: there’s no way to triangulate 2 or less points, so 0 is what we return
     2. Inductive case
        1. Note that we can split a polygon composed of points into two separate polygons by splitting the points on some index
        2. Thus we let:
           1. polygon composed of points
           2. polygon composed of points
        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 , we need to do the following:
           1. Figure out the efficient triangulations for all possible
           2. Draw the connecting triangles between and by connecting point to points and points
           3. Figure out the resulting cost of the above drawing by adding the costs of and , with the distance of the lines drawn between .
           4. Find the lowest overall total cost over all possible
           5. Note that the previous two steps are the English representation of , provided (i.e. not base case)
  4. Running Time Analysis
     1. Note the following ranges – the algorithm must do essentially 3 nested loops for each of the variables:
        1. , but since ’s minimum value is ,
        2. , but since ’s minimum value is 1 and ’s maximum value is n
     2. Since all of these ranges are size , their product must be . Thus, the algorithm runs in cubic time.

1. Profit-maximizing chocolate slicing

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

* 1. Semantic Array
     1. is the maximum profit able to be extracted from a chocolate bar with rows and columns
     2. Answer:
  2. Computational Array
     1. (base case 1)
     2. (base case 2)
  3. Array Equivalence
     1. There are three ways to split a chocolate bar:
        1. Don’t split it at all
        2. Split it on the row
        3. Split it on the column
     2. For these three cases, we could receive the following respective profits:
        1. , where our chocolate bar has rows and columns
        2. The maximum amount of profit we can extract out of the piece that goes to the row PLUS the maximum amount of profit we can extract out of the piece that goes beyond the row
        3. The maximum amount of profit we can extract out of the piece that goes to the column PLUS the maximum amount of profit we can extract out of the piece that goes beyond the column
     3. In order to maximize our profit, we have to take the largest of these three values
     4. Note that the previous two items essentially describe the computational array in plain English
  4. Running Time Analysis
     1. We note that the overall number of sub-problems is (area of the computational matrix). At each sub-problem, we have to consider possible row breaks and possible column breaks, plus a look-up for the profit of the intact bar. Thus, if the number of sub-problems is and the number of choices per sub problem is , our total runtime is .