# Problem 1

## a) Argue whether modifiedPrim always returns the correct answer by either arguing its correctness (if you think it's correct) or by providing a counterexample (if you think it's incorrect).

```
function modifiedPrim(G)
    T ← ∅
    H ← new heap containing only(ps, s), i.e., the pair of vertex s and its
associated positive integer using the integer as the key.
    while H /= ∅ do
        u ← H.removeMin()
        Add u to T, along with the edge to its parent(s has no parent)
        for (u, v) incident to u do
            if v /∈ T and v doesn't have a parent then
                if pu > pv then
                return null
            else
                    Set u to be the parent of v
                    Insert(pv, v) into H
    Set s to be the root of T

return T
```

Let $x$ be the next inserted vertex into the tree
**Invariant:** every newly inserted vertex ($x$) is only valid if it is increase increasing or equal to the last inserted element or (if it is decreasing) $x$ has a direct link to an element of lower or equal value, and will thus be ignored.

**Claim:** by keeping this invariant, the algorithm creates a monologue spanning tree.

The initial lowest number needs to be the root given, so no numbers ahead of that initial root can be less than the root. the lowest it can go is the root that are already in the tree.

The smallest value avaliable would always be inserted into the graph and therefore in the next iteration the value being inserted would be larger than the one just popped (because the one popped was the smallest, the next one must be the second smallest which means its larger than or equal to the previous) and therefore the values of the vertexes are always attempts to increase.

However, if a smaller number is encountered in a specific iteration then in a valid monologue that must mean that the vertex has a direct connection back to a previous vertex with the same value or a vertex with a lower value.

Otherwise, if it was never encountered before, then that number must mean that the graph had to forcefully decrease and thus is not a valid monologue

**Cases:** assume $x$ is a vertex being checked

1. $x$ is greater than or equal its parent
    1. $x$ is a valid value to a monologue and it's children will be checked in the next iteration. It has been marked with its parent.
2. $x$ is smaller than the parent
    1. if $x$ already has a parent that means that it was previously a valid part of the monologue seen in a different path. so it can be ignored
    2. if $x$ has not been seen before, then that means that it must have only larger elements as its parents (as the smaller or equal parents would have been seen by now), and thus that path must be decrimenting and is therefore not a monologue

### b) Argue whether modifiedDFS always returns the correct answer by either arguing its correctness (if you think it's correct) or by providing a counterexample (if you think it's incorrect).

```
function modifiedDFS(G)
    Initialize visited as in DFS
    Let s be the vertex with smallest integer.
    T ← ({s}, ∅)
    Set s to be the root of T

    modifiedDFSvisit(s)
    if T contains all vertices of G then
        return T
    else
        return null

function modifiedDFSvisit(u)
    Set visited[u] to true
    for (u, v) incident to u do
        if not visited[v] and pu ≤ pv then
            Add vertex v and edge (u, v) to T
            modifiedDFSvisit(v)
```

The modified DFS follows the similar lines of reasoning as the modified prims algorithm it sets the root to be the smallest integer, and attempts to visit its entire path from root to leaf.

In the DFS algorithm the vertex is ***only*** added to the spanning tree if it has never met the requirement of having the parent be of a lower value than the child. Consider a vertex that $v \in G$ but $v \notin T$. Because the DFS algorithm ensures that every vertex in the tree will be visited, the only condition where a vertex is not added to the tree, is if the vertex always lower than its parent in every path seen. Therefore, if one or more vertices never meet this condition, that means that at least one vertex must be decreasing from the parent to the child, which means that there is a decreasing path and thus is not a monologue.

## Problem 2

## a. Describe your algorithm in plain English.

```
1:def place_towers(houses):
2:  houses ← sort intervals in increasing starting range order
3:  houses ← sort by largest to smallest length if they have the same start
time
4:  towers ← stack containing the ranges, the initial value being [-INT_MIN,
INT_MAx]
5:  for house in increasing starting time order and do
6:      if house.start_range <= tower.peek().end_range
7:          towers.peek() ← house.start_range
8:          if house.end_range <= tower.peek().end_range
9:              towers.peek() ← house.end_range
10:     else if house.start_range > tower.peek().end_range
11:         tower ← house.start_range
12:         tower ← house.end_range
13:         towers.push(tower)
14: return towers.keys()
```

(with comments)

```
1:def place_towers(houses):
    # initialization
2:  houses ← sort intervals in increasing starting range order
3:  houses ← sort by largest to smallest length if they have the same start
time
4:  towers ← stack containing the ranges, the initial value being [-INT_MIN,
INT_MAx]
5:  for house in increasing starting time order and do
6:      furthest_tower = towers.peek()
        # if the current house is within range of the previous tower
7:      if house.start_range <= furthest_tower.end_range
            #move the tower forward to accomedate the new house
8:          furthest_tower() ← house.start_range
            #if you encounter a smaller end range however, update the tower's
end range
9:          if house.end_range <= furthest_tower.end_range
                # change the most recently added tower's range to be the end of
the new house (so the algo knows the extent of what it covers)
10:             furthest_tower ← house.end_range
        # if the current house is outside of the range of the old tower
11:     else if house.start_range > furthest_tower.end_range
            # create new tower with the range of the current house
12:         new_tower ← [house.start_range, house.end_range]
            #stack push, and because it is incrementing by houses start times,
whenever the value is looked at, it will always have the biggest start range
13:         towers.push(tower)
14: return towers.keys()
```

first sort the houses by their start range, (the lower part)
if there are ties, then sort by their lengths, the smallest, to largest. (this sorting isnt 100% necessary, but it helps with understanding how the towers work, it can be removed in an actual algo)

create a stack of towers with 2 values, initial value as int min and int max (essentially trying to cover the entire road). The start_range of the towers will be where it is actually placed, the end_range contains smallest value the tower can still cover (the end range of the house furthest back (by the way the towers are sorted))

iterate through the houses from smallest start range to largest

if the current house is within the range of the lastest add tower then move the tower to the starting position of the current house.

If the current house's ending range is smaller than the range of the most recent tower, then update the tower's range to be the house's ending range (essentially shrink the tower's ending range to the house's).

if the house's starting range exceeds the tower's end range, that means that the house is outside of the tower and a new tower need to be created. Create the tower with the houses start and end range and push it to the tower stack.

continue untill all the houses have been looped through

finally, loop through the stack and return the keys

# b) Prove the correctness of your algorithm.

**Observation:** place_towers never lets a house out of reach, whilst also including as many houses as it can with the same tower. Every tower has a range that keeps the most houses as possible, only once this range is exceeded, does it create a new tower to accomadate the a new house

**Claim:** place_towers is optimal.

**Proof:**

- length of towers = number of towers greedy algorithm allocates.
- A new tower is created *only* when a house exceed the ranges of a previously placed tower and cannot be moved, without isolating the previous house from cell service
- Because it is sorted by the starting time, every "out of range" house that cannot be reached by an existing tower, is caused by a house that has a later starting time than the house inside the tower.
- Therefore, it tries to include as many houses as possible by changing the tower's "valid" boundaries, as it tries to retain its furthest back house within reach for as long as it can, while also attempting to include new houses going forward. Only when a house is out of reach does the algorithm create a new tower.
- Key observation ⇒ all houses use >= d towers.

## c) Analyze the time complexity of your algorithm.

2: the sorting of houses by starting time using merge sort would take $O(n \cdot log(n))$ time
3: the sorting of houses by length (only for tiebreakers while retaining the starting time) using merge sort would take $O(n \cdot log(n))$ time
4: the creation of towers would take $O(1)$ time

5: looping through every house would take O(n) time
6: looking at the top of a stack in the peek() $O(1)$
7: the if statement inside takes $O(1)$
8: changing the value take $O(1)$
9: the if statement takes $O(1)$ time
10: changing the end value of the tower take $O(1)$ time
11: getting values of the tower range and the house ranges take $O(1)$ if statment takes $O(1)$
12: creation of new array takes $O(1)$
13: pushing to a stack takes $O(1)$
14: returning the towers positions takes $O(n)$ as it would need need to loop through the tower arrays and only retrieve the place where the tower is, tower.start_range or tower[i] [0]. retriving the value takes $O(1)$ time

total time = $O(n \cdot \log n) + O(n \cdot \log n) + O(n) \cdot O(1) + O(n)$
total time =$O(n \cdot \log n) + O(n \cdot \log n)$ (lower values than $O(n \cdot \log n)$ are not relavent)
total time = $2 \cdot O(n \cdot \log n)$
total time = $O(n \cdot \log n)$ (constants are removed)

# Problem 3

## a) Describe your algorithm in plain English.

**Divide:** Find the middle of the array length A and the middle of B and comparing the two together $O(1)$

**Recur Step:** recursively call the function according to the value of B[index],

- if A[index] is greater than the same B[index] then discard the right half (all bigger numbers and the current A[index]) and use the left half, till you get to the base case
- If A[index] is smaller than the B[index] then discard the left half (all smaller numbers than A[index]) and use the right half in the next iteration, until you get to the base case

**Base case:** if the array you are checking still fits within the bounds of the initial? if all numbers are exhausted return -1 to indicate that the perfect slice cannot be found

**Conquer:** assume the "recursion fairy" will give you the correct index needed, compare the index given A[i] == B[i] if its true return i otherwise keep recurring

```
#start and end begins as 0 and length of A - 1
1:def find_cake_slice(A, B, start, end):
2:  # base case
3:  if end >= start then
```

```
4:      # divide
5:      mid ← start + ⌊(end - start)/2⌋
6:
7:      # conquer
8:      if A[mid] == B[mid] then
9:          return mid
10:     #recurse
11:     if A[mid] > B[mid] then
12:         find_cake_slice(A, B, start, mid - 1)
13:     else A[mid] < B[mid] then
14:         find_cake_slice(A, B, mid + 1, end)
15:
16: else then
17:     return None
```

## b) Prove the correctness of your algorithm.

Since the array A is sorted increasing, and array B is sorted decreasing you can infer that increasing the index will increase the number value for A but will decrease number's the value in B.

i will be shorthand for `⌊n/2⌋` in this case being the middle point.

Let $x$ be an element where A[i] == B[i]

**Invariant:** If $x$ is in A before the divide step, then $x$ is in A after the divide step

- if A[⌊n/2⌋]>$x$, then $x$ must be A[0 to ⌊n/2⌋-1]
- if A[⌊n/2⌋]<$x$, then $x$ must be A[⌊n/2⌋+1 to n-1]

For example, if the algorithm finds the value in A[i] is smaller than B[i] this means that $x$ must be greater than A[i] but smaller than B[i]. Because A is sorted increasingly by taking A from `0 to i - 1` it will discard all values in A that are smaller than A[i]. Similarly, by taking the indexes at `0 to i - 1` it will also discard all values in B that are larger than B[i] (because B is sorted from largest to smallest). This will effectively shrink the pool of possible numbers the algorithm must look at, without eliminating the $x$ we are looking for. The same goes for when A[i] is greater than B[i].

Therefore, if $x$ exists in both A and B it will still exist after the divisions are run. If $x$ exists, then the algorithm will eventually shrink down to $x$ or, it will be found by checking the indexes before n = 1. The algorithm will continue to run until the array is empty, at which point the algorithm will know that $x$ doesnt exist.

## c) Analyze the time complexity of your algorithm.

**Divide step:** find middle in line 5 takes $O(1)$ and compare to the same middle inside B line 8, taking $O(1)$. both take $O(1)$ overall

**Recur step:** in each step the array would shrink in half $T(n/2)$ to fully reduce the array to 1 element would take $O(\log(n))$ time. The recur step only occurs once each iteration (its written twice, but in an if else statement)

**Conquer step:** returning the index after it matches the correct answer takes $O(1)$

The recurrence relation for this algorithm comes to
$T(n) = T(n/2) + O(1)$ for n > 2
$T(n) = O(1)$ for n = 1

Using the master theorem

**Case 1:** $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$ then
$T(n) = \Theta(n^{\log_b a})$

**Case 2:** $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ for $k \geq 0$ then
$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$

**Case 3:** $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $a \cdot f(n/b) \leq \delta f(n)$ for $\epsilon > 0$ and $\delta < 1$ then
$T(n) = \Theta(f(n))$

Using our recurrence relation
$T(n) = T(n/2) + O(1)$

$a = 1$
$b = 2$
$f(n) = O(1)$

first calculate $n^{\log_b a}$ to find if it is > than f(n) = to f(n) or < than f(n), and thus find the case
$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Choosing k = 0 we find that:
$f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$
$f(n) = \Theta(n^{\log_2 1} \cdot \log^0 n)$
$f(n) = \Theta(n^0 \cdot 1)$
$f(n) = \Theta(1 \cdot 1)$
$f(n) = \Theta(1)$

because f(n) is exactly the same, that would make this recurrence relation case 2
$f(n) = \Theta(n^{\log_b a}) \cdot \log^k n$
$n^{\log_b a} = O(1) = f(n)$

then we can use $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$ specified in case 2
$T(n) = \Theta(n^{\log_2 1} \cdot \log^{0+1} n)$
$T(n) = \Theta(n^0 \cdot \log^1 n)$
$T(n) = \Theta(1 \cdot \log n)$
$T(n) = \Theta(\log n)$

Therefore the algorthm occurs in $O(\log(n))$ time