Question 1: String breaking **[6+6+2]**

A string processing application that you are programming requires you to break-up a string into multiple parts. You are forced to use an existing library that has a string division function that divides a string into two parts at a given location. However, you find that the library function copies the string to be broken up before doing the division which takes time proportional to the length of the string. So, the order in which you divide the string changes the amount of time it takes to break-up the string. For example, suppose the original string length was 25 and the breaks had to happen after 4 characters, 8 characters and 10 characters in the original string. Then if you choose the ordering 4, 8, 10 for division then the time taken is proportional to 25 + 21 + 17 = 63. But if you choose the ordering 10, 8, 4 the time taken is proportional to 25 + 10 + 8 = 43. You have to devise a dynamic programming algorithm that given a string of length l and a sequence of m breaks finds the minimum time taken for the breaks. The breaks $l_1, \ldots, l_m$ are specified as the number of characters from the left end of the original string after which there should be a break.

Consider the following:

(i) b be the list of m break points in the original string
(ii) assume that the break is at breakpoint b_i
(iii) *bleft*(b, i): gives the list of break points to the left of the i^{th} break point that is break points from 1 to $i - 1$.
(iv) *bright*(b, i): gives the break points to the right of the i^{th} break point that is from $i + 1$ to m.

(a) Using the above information formulate a recurrence to compute the minimum time required to break the string of length l.
(b) Write a pseudocode (C/C++ details like header declaration, prototype declaration etc. may be omitted).
(c) Compute the runtime complexity of the algorithm.

Answer:

We can write the following naive recurrence for the division problem.
First note that if $m = 0$ then there is no break so time is 0. Similarly, if there is just 1 break point the time taken is always l irrespective of where the break point is located. These are our base cases.
For the recursive case let b be the list of m break points in the original string of length l and assume the break is at break point b_i.
Assume we have the following functions:
$bleft(b, i)$: gives the list of break points to the left of the i^{th} break point that is break points from 1 to $i - 1$.
$bright(b, i)$: gives the break points to the right of the i^{th} break point that is from $i + 1$ to m. Note that $bright(,)$ will have to return suitably modified indices since it will now be in a string of size $l - b_i$.
Let function $f(l, b)$ give the minimum time required for breaking a string of length l at break points b. The we have:

$$f(l,b) = \begin{cases} 0, & \text{if } m \text{ is } 0 \\ l, & \text{if } m \text{ is } 1 \\ \min\limits_{i \in 1..m} \left(f(b_i, bleft(b,i)) + f(l - b_i, bright(b,i)) + l, & m > 1 \end{cases}$$

The above recurrence breaks a string into two parts at every possible break point, adds the minimum time for each part and adds l to the result. And this happens recursively for each string. So, if we have a large number of breaks, say $O(l)$ breaks, we will end up

having a large number of overlapping computations. Actually, it looks at all possible permutations of break points at each stage to get the order giving the minimum time. We use memoization for a DP solution. We let MEMO be a $l \times l$ matrix where $MEMO[i][j]$ gives the minimum time taken for the substring between characaters i to j (both included). Changing the focus from breaks to characters in the string is the key to applying dynamic programming. We can define the new recurrence as follows where $f(i,j)$ gives the minimum time for a substring from i to j:

$$f(i,j) = \begin{cases} (j-i+1) + f(i,b) + f(b+1,j) & \text{if a break point } b \text{ exists between } i \text{ and } j \\ 0 & \text{if there is no break between } i,\ j \text{ - base case} \end{cases}$$

The answer we want is: $f(1,l)$. The algorithm looks as follows:

```
Initialize l × l array MEMO to ∞.
For all i, j such that j > i and there is no break between i and j set
MEMO[i][j] = 0.
This is a quick way to initialize large parts of the array.

function f(i,j) {
   if (f(i,j) ≠ ∞) return f(i,j)
   else {
     tmp = ∞
     for each b ∈ brks {
       if (b between i and j)
         tmp = min(tmp, j − i + 1 + f(i,b) + f(b+1,j))
       MEMO[i][j] = tmp //remember in MEMO
     }
   }
   return tmp
}
```

Note that each sub-problem is computed only once.

Answer:
The algorithm above fills up the MEMO array which is order $O(l^2)$ and it goes over each break. The number of breaks in the worst case can be $O(l)$ so complexity is $O(l^3)$.

Question 2: Non-duplicates **[5+5]**

You are given a sorted array of numbers where every value except one appears exactly twice; the remaining value appears only once. Design an efficient algorithm for finding which value appears only once. Here are some example inputs to the problem:

1 1 2 2 3 4 4 5 5 6 6 7 7 8 8
10 10 17 17 18 18 19 19 21 21 23
1 3 3 5 5 7 7 8 8 9 9 10 10

One can always solve this problem easily in O(n) where n is the number of elements in the array. The challenge is to bring it down to O(log n). Considering $n = 2k + 1$, design a divide-and-conquer style algorithm to solve the problem in O(log k). Formally prove that the algorithm that you have suggested is correct – correctness of the algorithm on the smaller inputs guarantees the correctness on larger inputs.

The key insight needed to solve this problem is the following: suppose you look at the pairs of elements at positions (0, 1), (2, 3), (4, 5), etc. All pairs that appear before the singleton element must consist of two copies of the same value. If we look at the pair containing the singleton value, then the first and second element of the pair will be different, with the singleton element at the first position. Importantly, if we look at any pair *after* the singleton element, the values in that pair will be different, since the pair containing the singleton will have "stolen" an element from the pair that appears after it, shifting everything down by one position.

Let's see how to formalize this into an algorithm. We can pick a pair of elements close to the middle of the the array and check whether the two elements there are equal or different. If they're equal, we can discard that pair and all pairs that come before it, since the singleton element must come after it. If they're different, we can discard the second element of that pair and everything that comes after it, since we know that the second element is part of a pair and that the singleton is either the first element of the pair or comes before it. Finally, when we get down to one element left, we know that element will be the singleton. We'd expect this to run in time $O(\log n)$, since we're discarding about half the elements on each iteration. Let's begin by formalizing the algorithm:

> **Algorithm:** Let A be our array and let its length be $n = 2k + 1$ (since it consists of k pairs and one singleton). If $n = 1$, return A[0]. If $n > 1$, compare A[2⌊k / 2⌋] and A[2⌊k / 2⌋ + 1] (using zero-indexing). If the elements are equal, recursively apply this algorithm to the subarray beginning at position 2⌊k / 2⌋ + 2 and extending to the end. Otherwise, recursively apply this algorithm to the subarray starting at the beginning of the array and extending to 2⌊k / 2⌋, inclusive.

Now that we have a formal version of the algorithm, we need to prove that the algorithm works correctly. This is a lot trickier than it might initially appear to be. In order to show correctness, we need to show that we can determine in which half of the array the singleton element appears simply by looking at the elements at positions 2⌊k / 2⌋ and 2⌊k / 2⌋ + 1. Intuitively:

1. If the singleton appears after positions 2⌊k / 2⌋ and 2⌊k / 2⌋ + 1, then all the elements before the singleton would be paired up correctly. Therefore, A[2⌊k / 2⌋] = A[2⌊k / 2⌋ + 1], so we can discard the first half of the array.
2. If the singleton appears at or before position 2⌊k / 2⌋, then all the elements after the singleton would be mismatched. Therefore A[2⌊k / 2⌋] ≠ A[2⌊k / 2⌋ + 1], so we can discard the second half of the array (but not A[2⌊k / 2⌋], since it might be the singleton).

Based on these observations, we might want to try to prove the following lemma:

> *Lemma:* A[2m] ≠ A[2m + 1] iff the position of the singleton (denoted s) satisfies $s \leq 2m$.

For now, let's hold off on proving this statement (we're pretty sure that it's true). Instead, let's see if we can prove that the algorithm is correct under the assumption that this lemma holds. If it doesn't, then we probably shouldn't waste our time trying to prove it!

> **Useful Tip #1:** Before proving any lemmas, confirm that you can prove that your algorithm is correct under the assumption that those lemmas are true. If so, go back and prove the lemmas. If not, discard or modify the lemmas.

Since our algorithm is inherently recursive, we are probably best off trying to prove it correct with induction, showing that the correctness on smaller inputs guarantees correctness on larger inputs. The algorithm is supposed to find the singleton element, so we should prove this is so:

Theorem: Given an array of size $2k + 1$, the algorithm returns the singleton element.

Proof: By induction on k. As a base case, when $k = 0$, the array has length 1 and the algorithm will return the only element, which must be the singleton. For the inductive step, assume that for some k that the claim holds for all $0 \leq k' < k$. We will prove it holds for k.

If $A[2\lfloor k / 2 \rfloor] = A[2\lfloor k / 2 \rfloor + 1]$, then by our lemma, we have $s \geq 2\lfloor k / 2 \rfloor + 1$. Because $A[2\lfloor k / 2 \rfloor + 1]$ is duplicated, this means $s \geq 2\lfloor k / 2 \rfloor + 2$. In this case, our algorithm recursively invokes itself on the subarray starting at position $2\lfloor k / 2 \rfloor + 2$. This subarray contains $2k + 1 - 2\lfloor k / 2 \rfloor - 2 = 2(k - \lfloor k / 2 \rfloor - 1) + 1$ elements. Therefore, by the inductive hypothesis, the recursive call returns the singleton element, so the algorithm returns the singleton element.

Otherwise, $A[2\lfloor k / 2 \rfloor] \neq A[2\lfloor k / 2 \rfloor + 1]$, so by the lemma, $s \leq 2\lfloor k / 2 \rfloor$. The algorithm then recursively invokes itself in the subarray ending at position $2\lfloor k / 2 \rfloor$, which has $2\lfloor k / 2 \rfloor + 1$ elements in it. Therefore, by the inductive hypothesis, the recursive call returns the singleton element, so the algorithm returns the singleton element. This completes the induction. ■

Now that we have this theorem in tow, we should come back to prove that the lemma is true. As a reminder, our lemma is

Lemma: $A[2m] \neq A[2m + 1]$ iff the position of the singleton (denoted s) satisfies $s \leq 2m$.

This is a biconditional, so we can try thinking about how we'd show each direction of implication.

Let's start with the forward direction: if $A[2m] \neq A[2m + 1]$, then $s \leq 2m$. Why is this true? Intuitively, if these elements aren't paired up, then the singleton element has to be before them or equal to one of these two elements. In the latter case, this is easy to show. In the former case, we would somehow need to argue that the singleton had to have come earlier in order to offset the elements. This is true, but difficult to prove.

Let's try a different tactic: the contrapositive of this statement is

$$\text{If } s > 2m, \text{ then } A[2m] = A[2m + 1]$$

In other words, if the singleton comes after position $2m$, then $A[2m]$ and $A[2m + 1]$ would have to be paired with one another. The intuition here is a bit easier to work with: if the singleton appears after position $2m$, then everything before it must be paired up. We could easily prove that this is true using induction, since

- The first two elements must be paired up, since neither is the singleton.
- If the first $2r$ elements (where $2r \leq 2m < s$) are paired up and the element at position at $2r$ isn't the singleton, it has to be paired with the element at position $2r + 1$ or $2r - 1$. By the IH, the element at position $2r - 1$ is already paired up, so it has to be paired up with the element at position $2r + 1$.

So now we have to think about the reverse direction: if the singleton appears at or before position $2m$, then $A[2m] \neq A[2m + 1]$. We can prove this in more or less the same way that we proved the previous part: use induction to show that if the singleton is at some position $2t$, then all pairs after position $2t$ will be mismatched. Based on this discussion, we arrive at this formal proof:

Proof: We prove both directions of implication.

(⇒) By contrapositive; we will show that if $s > 2m$, then $A[2m] = A[2m + 1]$. To do this, we prove the stronger claim that $A[2r] = A[2r + 1]$ for all $0 \leq r \leq m$ by induction on r. As a base case, when $r = 0$, A[0] can't be the singleton ($0 \leq 2m < s$), so A[0] = A[1]. Now suppose that for some r that the claim is true for all r' satisfying $0 \leq r' < r \leq m$. We will prove the claim is true for r. We know $A[2r]$ is not the singleton, since then $s = 2r \leq 2m$, contradicting that $s > 2m$. Therefore, $A[2r]$ must be equal to an adjacent array element. We cannot have $A[2r] = A[2r - 1]$ or then $A[2(r - 1)] \neq A[2(r - 1) + 1]$, so we must have $A[2r] = A[2r + 1]$. Thus the claim holds for r, completing the induction.

(⇐) Suppose $s \leq 2m$. We claim that s must be even; if s were odd (say, $s = 2t + 1$), then we have $A[2t] \neq A[2t + 1]$, and by (⇒) this would mean $s \leq 2t < 2t + 1$, a contradiction. So s is even; let $s = 2t$. Then since all elements after the singleton must be paired, a quick inductive argument shows that $A[2t + 2k] \neq A[2t + 2k + 1]$ for all $k \geq 0$. Setting $k = m - t$ completes the proof. ■

Question 3: Complexities **[1+1+1+1+1+1]**

Consider the following three algorithms:

Algorithm 1 solves problems of size N by recursively dividing them into 2 sub-problems of size N/2 and combining the results in time c (where c is some constant).

Algorithm 2 solves problems of size N by solving one sub-problem of size N/2 and performing some processing taking some constant time c.

Algorithm 3 solves problems of size N by solving two sub-problems of size N/2 and performing a linear amount (i.e., cN where c is some constant) of extra work.

Formulate the recurrence relation in each case and obtain the solution. For each of the following algorithms, pick which of the above classes of algorithms (1, 2, or 3) applies to that algorithm: (i) Mergesort (ii) Binary search in a sorted array (iii) Quicksort (if partitioning always divides the array in half).

This was a free gift!

Question 4. Shortest Cycle in a weighted directed graph **[6]**

Describe an algorithm to find the length of the shortest cycle in a weighted directed graph in $O(n^3)$ time. Assume that all the edge weights are positive.

Solution Sketch: Use Floyd Warshall's algorithm but also update the diagonal entries now (initialize with infinity/int_max). At the end, the smallest diagonal entry will correspond to the shortest cycle in the graph.

Common Mistakes: Applying DFS / BFS / Prim's Algorithm / Bellman Ford, etc. This would not give you the shortest cycle in $O(n^3)$ time.

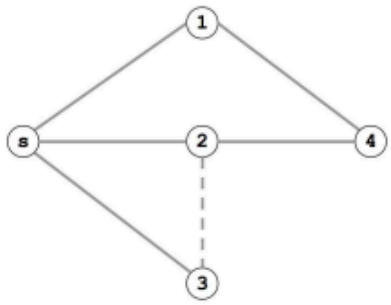
For instance, if you want to solve using traversal, here is an algorithm for unweighted graphs.

The key idea is that a shortest cycle is comprised of a shortest path between two vertices, say v and w, that does not include edge v-w, plus the edge v-w. We can find the shortest such path by deleting v-w from the graph and running breadth-first search from v (or w).

```
For each edge v-w
   - Form a graph that is the same as G, except that edge v-w is removed.
   - Find the shortest path dist(v, w) from v to w using BFS.
   - Compute dist(v, w) + 1, which corresponds to the cycle consisting
     of the path from v to w, plus the edge v-w.
   - If this is shorter than the best cycle found so far, save it.
```

We run BFS E times and each run takes $O(E + V)$ time. The overall algorithm takes $O(E(E + V))$ time.

Note that if you run BFS from s and stop as soon as you revisit a vertex (using a previously ununused edge), you may not get the shortest path containing s. For example, in the following graph, BFS might consider the edges s-1, s-2, s-3, 1-4, 2-4, thereby finding the cycle s-1-4-2-s (instead of the shorter cycle s-2-3).



Similarly, if you use Dijkstra's algo, the complexity would be O(E(ElogV))
https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/

Question 5. DFS Edge Classification **[6]**

Suppose you are performing a DFS traversal over an undirected graph, and you get the DFS tree. Would you still get all the four edge types or a subset of these? If all the edge types are not possible, which ones would you get? Justify your answer.

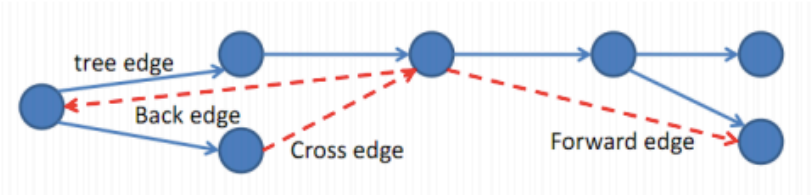
Solution Sketch: DFS tree of an undirected graph contains only two types of edges: **tree edges** and **back edges**. Forward edges and cross edges will not be there. [Provide justification-using contradiction – e.g.,

assume there is a forward edge, etc.]
3 marks for showing that forward edges would not exist.
3 marks for showing that cross-edges would not exist.

DFS Edge Classification

The edges we traverse as we execute a depth-first search can be classified into four edge types. During a DFS execution, the classification of edge (u, v), the edge from vertex u to vertex v, depends on whether we have visited v before in the DFS and if so, the relationship between u and v.

1. If v is visited for the first time as we traverse the edge (u, v), then the edge is a **tree edge**.
2. Else, v has already been visited:
   (a) If v is an ancestor of u, then edge (u, v) is a **back edge**.
   (b) Else, if v is a descendant of u, then edge (u, v) is a **forward edge**.
   (c) Else, if v is neither an ancestor or descendant of u, then edge (u, v) is a **cross edge**.



After executing DFS on graph G, every edge in G can be classified as one of these four edge types. We can use edge type information to learn some things about G. For example, **tree edges** form trees containing each vertex DFS visited in G. Also, G has a cycle if and only if DFS finds at least one **back edge**. Note that undirected graphs cannot contain **forward edges** and **cross edges**, since in those cases, the edge (v, u) would have already been traversed during DFS before we reach u and try to visit v.

Common Mistakes:

1. **DFS tree** would only contain tree edges and no other edges. Isn't that obvious? What is there to ask in the question in that case? DFS tree is only used to be able to name the edge types.
2. Forward and back edges are indistinguishable, so only one of them (back) edge will exist → Then, why not forward edge?

Question 6. Balanced Binary Trees **[3+3]**

A family of binary trees is called balanced if every tree in the family has height *O(lg n)*, where *n* is the number of nodes. For the following properties, prove or disprove that the family of binary trees satisfying that property leads to balanced trees. Note that this is an asymptotic property.

(a) Every node of the tree is either a leaf or it has two children

Solution: Not balanced. Counterexample is a right chain with each node having a leaf hanging onto its left.

Common Mistakes:

1. Giving a counterexample with a small n and h. (E.g., n=9/11) and showing that h is not *logn*. How can one talk about O(logn) for small values of *n* and *h*? Didn't the question clearly say that this is an asymptotic property? You have to give a counter-example for a generic *n* and *h*. By this logic (small n and h), one can even prove that AVL trees are not height-balanced.
2. Showing that a tree that satisfies this property does not satisfy AVL property!! AVL trees are just

one particular family of height-balanced BSTs. The question was giving several other properties and asking if this would lead to height balancing. So, showing that this is not the same as AVL doesn't prove anything except that this is not the same as AVL trees!

(b) The average depth of a node is *O(lg n)*. The depth of a node is the number of edges along the path from root of the tree to the node.

Solution: No.

Consider a tree with $n - \sqrt{n}$ nodes organized as a complete binary tree, and the other $\sqrt{n}$ nodes sticking out as a chain of length $\sqrt{n}$ from the balanced tree. The height of the tree is $\lg(n - \sqrt{n}) + \sqrt{n} = \Omega(\sqrt{n})$, while the average depth of a node is at most

$$\begin{aligned} & (1/n)\left(\sqrt{n} \cdot (\sqrt{n} + \lg n) + (n - \sqrt{n}) \cdot \lg n\right) \\ = \; & (1/n)(n + \sqrt{n}\lg n + n\lg n - \sqrt{n}lgn) \\ = \; & (1/n)(n + n\lg n) \\ = \; & O(\lg n) \end{aligned}$$

Thus, we have a tree with average node depth $O(\lg n)$, but height $\Omega(\sqrt{n})$.

Common Mistakes: Proving that this is balanced

Question 7. Binary Search Trees with the worst height **[2+4]**

You are given a sequence of integers $a_1, a_2, \ldots, a_n$ in an array. You need to decide whether inserting these integers in that sequence leads to a height of *n-1* of the binary search tree (worst possible height).

(a) If you actually build up the tree, what would be the time complexity in the worst case?

Solution: $O(n^2)$

(b) Propose an efficient algorithm for this. Since you need to read *n* elements, you can not do better than *O(n)*.

$\Rightarrow$ In order that we get a chain of length $n-1$, we need each non-leaf node to have only one child. This limits the allowed values of a_i given that $a_1, a_2, \ldots, a_{i-1}$ have already resulted in a binary search tree of height $i-2$. For example, if $a_2 < a_1$, then a_2 is inserted as the left child of the root a_1. But then, the root cannot have a right child, that is, none of $a_3, a_4, \ldots, a_n$ can be larger than a_1. The following pseudocode implements these observations.

- 1. If $(n \leq 2)$, return true.
- 2. Initialize lower limit $= -\infty$ and upper limit $= +\infty$.
- 3. For $i = 2, 3, 4, \ldots, n$, repeat:
- 4. If $(a_i <$ lower-limit) or $(a_i >$ upper limit), return false.
- 5. If $(a_i < a_{i-1})$, set upper limit $= a_{i-1}$.
- 6. If $(a_i > a_{i-1})$, set lower limit $= a_{i-1}$.

Common Mistakes:

1. That the elements have to be in sorted (ascending or descending) order, and giving an algorithm to find that. The elements need not be in sorted order, see for instance: 3, 10, 8, 9 → Inserting these in sequence will lead to a height of 3.
2. Actually inserting the elements in a tree → But that is what the part (a) asked. The worst case would be $O(n^2)$ if you do so.

Question 8. Minimum Spanning Tree **[6]**

Suppose that you are given a graph $G = (V, E)$ and its minimum spanning tree T. Suppose that from G, we delete one of the edges (u, v) in T and let G' denotes this new graph. Assuming that G' has a minimum spanning tree T', describe an efficient algorithm for finding T'. Thus, using an MST algorithm over G' is not a good option.

Solution: Since (u,v) in T has been deleted, T has 2 connected components. Idea is to find the minimum weight edge between the two components. To do that, we need to **first find the two components**. Use BFS from u or v to find one of the components. Other component consists of all other vertices.

Common Mistakes:

1. Did not give an algorithm to find the two components.
2. Used Prim's Kruskal's algorithm → This would be inefficient. The above algo can help you to solve in O(n+e). O(n) for detecting the two components, and O(e) to find the minimum weight edge between the two components. Of course, you may have to do some pre-processing but that will only take O(n+e).