Problem 1

1. D

2. 3 and 4 are true.

3. k = 5

4. f(n) = O(g(n))

5. 1536

6. 8 swaps

7. [1,2,3,4,5,6,7,8,9]
8.
   (a) Height of a Node
   The height of a node in a binary tree is the number of edges on the longest downward path between that node and a leaf. It can be seen as the height of the tree rooted at that node.

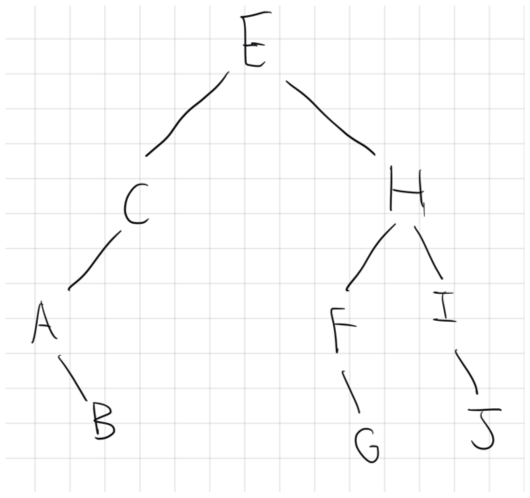
   (b) Depth of a Node
   The depth of a node is the number of edges from the tree's root node to the node. It measures how far a node is from the root. The concept of depth is relative to the root of the tree.

9. a) **Pre-order Traversal** (visit the root, traverse the left subtree, traverse the right subtree): **a, b, d, e, c, f, h, l, m, q, i, g, j, n, o, k, p**

b) **In-order Traversal** (traverse the left subtree, visit the root, traverse the right subtree): **d, b, e, a, l, h, q, m, f, i, c, n, j, o, g, p, k**

c) **Post-order Traversal** (traverse the left subtree, traverse the right subtree, visit the root): **d, e, b, l, q, m, h, i, f, n, o, j, p, k, g, c, a**

10. The original balanced binary search tree that gave the provided in-ordered and post-ordered outputs can be represented as follows:



Problem #2

a. Worst-Case Scenario Analysis:

1. **Outer Loop (for i):** This loop iterates from 1 to n, suggesting it has the potential to iterate n times over the array. In the worst-case scenario (such as when the array is sorted in reverse order), it's prepared to iterate through the entire array.

2. **Inner Loop (for j):** For each iteration of the outer loop, the inner loop iterates from 1 till (n – i), reducing the number of comparisons in each subsequent iteration since the last element(s) are already sorted and don't need to be checked again.

3. **Swapped Flag**: In the worst-case scenario, at least one swap will be performed during each iteration of the inner loop until the array is fully sorted. This means the early stopping condition (swapped = False) will not trigger until the array is actually sorted, which, in the worst case, does not happen until all iterations are complete.

Worst-Case Running Time Calculation:

In the worst case, where the array is in reverse order or requires maximum swaps, the swapped flag does not provide an early exit until the sorting is complete. Therefore, the analysis focuses on the maximum number of comparisons and swaps:

Comparisons: The total number of comparisons in the worst case is the sum of an arithmetic series from 1 to (n-1), which is n*(n – 1)/2.
Swaps: In the worst case, the number of swaps will also approach the number of comparisons, so it's similarly n * (n – 1)/2

The worst-case time complexity remains (O(n^2)). This is because, in the worst-case scenario, the algorithm still needs to compare and potentially swap elements through nearly the entire array, similar to the basic bubble sort algorithm.

b. Proof of Asymptotic Improvement:
1. Assumption:
Assume that the best-case running time of early-stopping Bubble Sort is not asymptotically better than that of basic Bubble Sort. This suggests they both share the same complexity class for their best-case scenarios, which is O(n^2) for basic Bubble Sort.

2. Implications:
If our assumption is true, it implies that for any input size n, there exists a constant c such that the running time of early-stopping Bubble Sort is at least c·n^2. This is

because the basic Bubble Sort, without any optimizations, has a running time characterized by n^2 behavior even in the best case.

3. Contradiction:
However, consider an already sorted array, which is the best-case scenario for sorting algorithms. The early-stopping Bubble Sort recognizes that the array is sorted much sooner; specifically, it only needs to pass through the array once without making any swaps, leading to a running time of O(n). This is because it checks for the absence of swaps to conclude early that the list is already sorted. No matter how small the constant c is, c·n^2will always be greater than or equal to n^2, and thus, it will always be larger than the linear time, O(n), required by early-stopping Bubble Sort in this scenario.

4. Conclusion:
The initial assumption leads us to a contradiction, indicating that the best-case running time of early-stopping Bubble Sort is indeed asymptotically better than that of the basic Bubble Sort.

Generalization:

This proof illustrates a fundamental principle in algorithm optimization: introducing a condition that allows an algorithm to terminate early can significantly improve its performance in best-case scenarios. While the worst-case scenario might not see improvement (both versions of Bubble Sort have a worst-case complexity of O(n^2)), optimizing for best-case performance can lead to more efficient algorithms in practical situations where the worst-case scenario is rare or less relevant.

Mathematical Proof:

The contradiction arises from our understanding that a linear function, O(n), grows slower than any quadratic function, O(n^2), as n becomes large. This is a fundamental result from calculus and asymptotic analysis, demonstrating that if an algorithm can achieve O(n) performance under any circumstances where another algorithm requires O(n^2), the former is asymptotically more efficient in those circumstances.

Thus, early-stopping Bubble Sort, by leveraging the additional information gained during its execution (whether any swaps were made), demonstrates a clear asymptotic improvement in its best-case running time over the basic Bubble Sort, moving from a quadratic to a linear complexity class.

c. Initialization and Construction of R**:**

- **Constructing** R **from** A: This process involves creating an array R such that each element R[i] is assigned the value of A[n−i+1], effectively reversing the order of A. Given that Ais reverse-sorted, R will be sorted in ascending order after this single pass-through, which requires theta(n) time. This construction is straightforward but crucial, as it sets up R as initially sorted.

First Iteration on A:

- **Moving the Smallest Element**: The early-stopping Bubble Sort begins its process on A by comparing adjacent elements and swapping them if necessary, to move the smallest element (initially at the end of A) to the first position (A[1]) through n−1swaps. This operation also takes theta(n) time, as it systematically moves the smallest element to its correct position, iterating through the entire length of A.

First Iteration on R**:**

- **Immediate Early Stopping**: When the sorting process starts on R, the algorithm quickly determines that R is already in ascending order. This is due to the early-stopping mechanism, which checks for the absence of any swaps in an iteration. Since R requires no swaps, the algorithm concludes its operation on R in theta(1) time, demonstrating the efficiency of early stopping in the best-case scenario.

Termination Check:

- **Identifying Sorted Arrays**: After the first iteration, the algorithm evaluates both A and R to determine if they are sorted. R, being unchanged and initially constructed as sorted, is immediately recognized as such, prompting the algorithm to terminate its operations.

Conclusion and Generalization:

- **Overall Running Time**: The total running time for applying an early-stopping Bubble Sort on this setup is dominated by the theta(n) time required to construct R and the theta(n) time needed for the first iteration on A, resulting in an overall running time of theta(n).

Mathematical Justification:

The effectiveness of early-stopping Bubble Sort in this scenario illustrates an important principle in algorithm design: optimizing for specific cases can significantly reduce running time. In this case, recognizing a sorted array allows the algorithm to minimize unnecessary operations.

For the entire array and the specified durations (1, 2, 3, 4), this process underscores the ability of early-stopping Bubble Sort to adapt its operations based on the array's initial state. In the best-case scenario (a sorted array), the algorithm demonstrates linear time complexity (theta(n)), a significant improvement over the quadratic time complexity (theta(n^2)) of traditional Bubble Sort without early stopping.

d. Analysis for Upper Bound O(n^2)

Construction of R

The construction of R from A requires n operations, which is linear and therefore O(n). However, this step is not the dominant factor in determining the algorithm's worst case time complexity.

Iterative Sorting of A and R:

The worst-case scenario for bubble sort (and similarly for early stopping bubble sort when early stopping does not occur) requires performing comparisons and swaps for nearly every pair of elements in the array. Specifically:
- For each iteration on A, in the worst case, (n-1) comparisons are needed for the first pass, (n-2) for the second, and so on, down to 1 comparison for the last pass, leading to n(n-1)/2 comparisons and potentially swaps.
- Similarly, the same number of comparisons and swaps would be required for R in the worst case.

Total Worst-Case Operations:

each array can contribute up to n(n-1)/2 operations in the worst case, making the sorting operations for both A and R collectively contribute to O(n^2) complexity.

Conclusion:

The maximum time complexity of the forward-backward bubble sort is dictated by the square of the number of comparisons and swaps needed to order both A and R under the most challenging conditions. Given that these operations are the primary contributors to the algorithm’s execution time, and their cumulative count is capped by a quadratic function of n, it follows that the forward-backward bubble sort’s worst-case time complexity is O(n^2)

e. To prove that the worst-case running time of the forward backward bubble sort is ohm(n^2) it is necessary to demonstrate that there exists at least one instance of an array A[1...n] for which the early stopping bubble sort requires ohm(n^2) time complexity on both the array A itself and its reverse R[1...n] where R[i]=A[n−i+1] for each i

Constructing Array A[1...n]
Let’s describe an array A[1...n] that is nearly sorted but has the smallest element at the end of the array:
- A=[2,3,4,...,n,1]

This array is almost in ascending order, except that the smallest element is positioned at the very end. For such an array, the early-stopping-bubblesort algorithm will not be able to terminate early because it needs to move the smallest element from the end of the array to the beginning, requiring n−1 swaps in the first iteration, n−2 in the second, and so on, until the smallest element is correctly placed at A[1] This process results in a total number of operations that is the sum of the first n−1 positive integers, which is n(n−1)/2leading to a ohm(n^2) time complexity.

Constructing the Reverse Array R[1...n]
For the constructed A[1...n], its reverse R[1...n] would be:

- R=[1,n,n−1,...,4,3,2]

This configuration of R places the largest element (which is n) at the beginning, followed by a reverse-sorted arrangement of the remaining elements. Running early-stopping-bubble sort on R requires moving the largest element to the end, requiring a similarly exhaustive series of comparisons and swaps as with A. The process involves moving the largest element n through each position until it reaches R[n] accumulating a comparable number of operations to what was described for A, thereby also resulting in a big ohm(n^2) time complexity for sorting R.

Problem #3

a.

Recurrence Relation and Base Case:

The recurrence relation for the time complexity of Algorithm B is:

- **T(n) = T(n-1) + theta(1)** (Represents the recursive call with reduced input size and constant-time work outside the recursion)
- **T(0) = theta(1)** (The base case where the input array is empty)

Change of Variable Method:

To align with the Master Theorem's form, let's introduce a change of variable: k = n - 1. This transforms the recurrence into:

- **T(k + 1) = T(k) + theta(1)**

Iterating this recurrence from the base case up to 'n' we get:

- T(1) = T(0) + theta(1)
- T(2) = T(1) + theta(1) = T(0) + 2 * theta(1)
- ...
- T(n) = T(n-1) + theta(1) = T(0) + n * theta(1)

Therefore, **T(n) = theta(n)**, demonstrating a linear time complexity.

Master Theorem Application:

While the original recurrence doesn't directly fit the Master Theorem's form, let's adapt it:

1. **Modified Recurrence:** Consider the case where A[1] >= 0. The recurrence becomes T(n) = T(n-1) + theta(1). Let's represent the work outside the recursion as f(n) = theta(1). This gives us:
   - T(n) = 1 * T(n/1) + f(n)
2. **Comparison:** Comparing with the Master Theorem's form (T(n) = aT(n/b) + f(n)):
   - a = 1
   - b = 1
   - f(n) = theta(1)
3. **Applying the Theorem (Case 2):** Since f(n) = theta(n ^ (log a base b)) = theta(n ^ 0) = theta(1), we fall into Case 2 of the Master Theorem. This implies:
   - T(n) = theta(n ^ (log a base b) * log n) = theta(n * log n)

Important Note: The Master Theorem analysis above assumes that we are consistently in the case where A[1] >= 0. In reality, the behavior would depend on the data distribution within the input array.

Conclusion:

Both the change of variable method and a modified application of the Master Theorem demonstrate that the overall running time of Algorithm B is in the order of theta(n), indicating linear time complexity. However, the Master Theorem result should be interpreted with the understanding that its applicability relies on the assumption of a specific input distribution.

b. Recurrence Relation for Algorithm C
Algorithm C divides the array into two equal parts (if n is even, or nearly equal if n is odd), and then recursively processes each part. This gives us the following recurrence relation:

T(n)=2T(n/2)+c

The base cases are:

- T(0)=c1, when n=0,
- T(1)=c2, when n=1 and A[1]≥0,
- T(1)=c3,. when n=1 and A[1]<0.

For simplicity, let's assume all base cases take constant time, T(1)=theta(1) and focus on the recursive case.

Guess and Confirm (Substitution Method)

Guess: The running time of Algorithm C is theta(n)

Confirm via Induction:

To prove our guess using induction, we must show that if T(k)=theta(k)for all k<n, then T(n)=theta(n).

Base Case for Induction: The base of our induction is trivially true since for n=1, T(1)=c2 or c3, which is theta(1).

Assumption: Assume for some k<n, T(k)=theta(k) We need to show that T(n)=theta(n) holds.

Given our recurrence relation: T(n)=2T(n/2)+c

Inductive Step:

By our inductive hypothesis, T(n/2) = theta(n/2) Substituting this into our relation, we get: T(n)=2theta(n/2) + c = theta(n) + c

Since c is a constant, adding c does not change the asymptotic behavior, and we have: T(n)=theta(n)

This confirms our guess that T(n) = theta (n) is indeed correct.

c. Algorithm D Recurrence Relation

The recurrence relation for Algorithm D, based on its division of the problem into segments, is:

T(n)=2T(n/4) + T(n/2) + c

where c represents the constant time for the operations outside of the recursive calls.

Building the Recursive Tree

To understand the running time, we'll construct a recursive tree, breaking down the problem at each level according to the recurrence relation.

1. **Level 0 (Root):**
   - The work done is c, which is the base operation for splitting the array.
2. **Level 1**:
   - Here, the problem is divided into three parts: two sub-problems of size n/4 and one sub-problem of size n/2. The work at this level is still c, applied to each part, but we're focusing on the division of the problem size.
3. **Subsequent Levels**:
   - Each sub-problem of size n/4 further divides into smaller sub-problems of sizes n/16 and n/8, and similarly, the sub-problem of size n/2divides into n/8and n/4.

Work at Each Level

- **Level 1**: The work is c×3=3c (ignoring the exact sizes for simplicity).
- **Level 2**: Each sub-problem at level 1 generates further sub-problems. The total work remains proportional to n, as we're essentially covering the whole array over the levels.

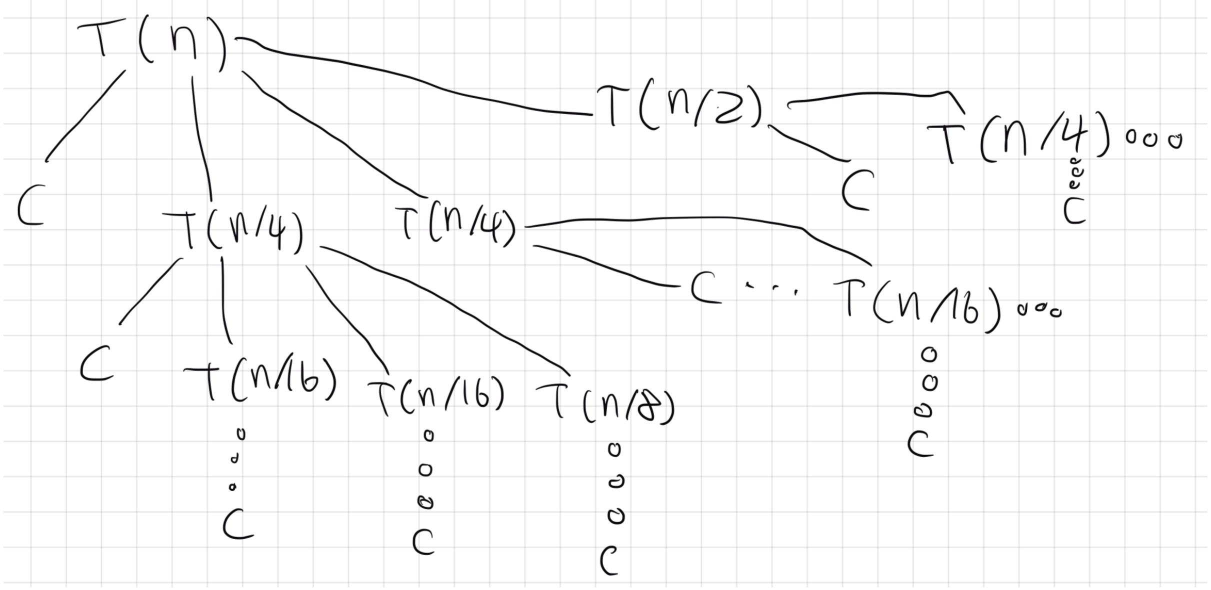
Total Work Across All Levels

- Given the division into sub-problems, the work at each level can be considered to contribute a constant amount to the total work proportional to the original problem size n.
- The depth of the tree suggests a logarithmic number of levels, but because the work at each level sums to O(n) the depth of the tree does not multiply the total work beyond O(n).

Justification of the Recursive Tree Method

1. **Visual Representation**: The recursive tree method provides a clear visual representation of how the algorithm divides the problem into smaller sub-problems at each step of the recursion. This makes it easier to understand the division pattern and how work accumulates across different levels of recursion.
2. **Accumulation of Work**: By breaking down the problem at each level, we can accurately calculate the total work done by the algorithm. This includes both the work of dividing the problem into smaller parts (represented by the constant time c*c*) and the recursive work on sub-problems.
3. **Depth and Work Analysis**: The method allows for an examination of both the depth of the recursion and the amount of work done at each level. This dual analysis is crucial for determining the overall time complexity of the algorithm.

Visual Representation:



Conclusion

The recursive tree for Algorithm D shows us that even though the problem isn't split evenly, the total amount of work at each tree level is still proportional to n. Plus, the tree's depth increases in a logarithmic fashion. This tells us that the running time isn't really affected by how deep the recursion goes. Instead, it's the linear work done at every tree level that matters. So, we can say that Algorithm D's running time is Theta(n).

d. All three algorithms (A, B, and C) solve the problem of **counting the number of negative elements in an array**. Here's a breakdown of how they each approach the problem:

Algorithm A

- **Recursive Approach:** Iterates through the array. If a negative element is found, it adds one to the count and recursively calls itself on the remaining subarray.
- **Focus:** Identifying the presence of negative numbers rather than their exact positions.

Algorithm B

- **Divide and Conquer:** Splits the array into two halves and recursively counts negative elements in each half. The sub-counts are then added together.
- **Focus:** Breaking down the problem to process parts of the array.

Algorithm C

- **Multi-way Divide:** Splits the array into three smaller subarrays (first quarter, middle half, last quarter) and recursively counts negative elements in each. The results are combined.
- **Focus:** More granular division of the problem.

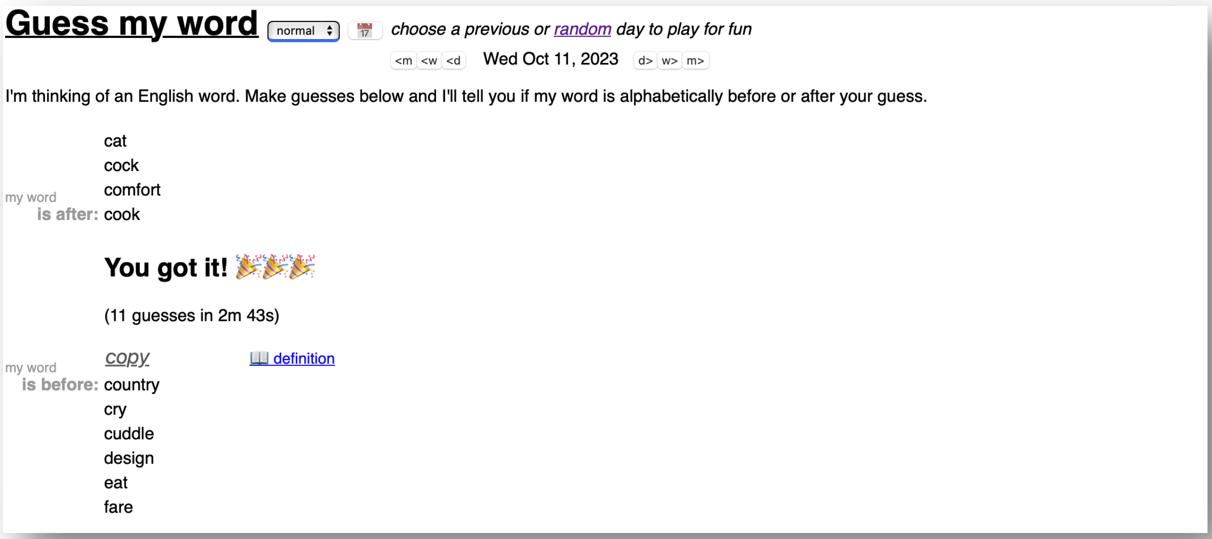
Why They Are All O(n)

Despite their different strategies, all algorithms have a linear time complexity for the following reasons:

- **Worst Case:** In the worst-case scenario, with no negative numbers, each algorithm must still examine all elements in the array once.
- **Bound on Work:** The total work done by recursion is proportional to the number of elements. This leads to a linear relationship between the size of the array and the time taken.

Problem #4

a.



Guess my word normal choose a previous or random day to play for fun

<m <w <d Wed Oct 11, 2023 d> w> m>

I'm thinking of an English word. Make guesses below and I'll tell you if my word is alphabetically before or after your guess.

cat
cock
comfort
my word is after: cook

You got it! 🎉🎉🎉

(11 guesses in 2m 43s)

copy definition

my word is before: country
cry
cuddle
design
eat
fare

b. To identify the secret word in a dictionary with exactly 2^k −1words in at most k guesses, we can use a binary search algorithm. The binary search algorithm works by repeatedly dividing in half the portion of the list that could contain the secret word, then selecting the midpoint as your next guess, effectively reducing the search space by half each time.

Algorithm

1. **Initialize**: Start with the entire dictionary as your search space. Set **low** as the first index of the dictionary (1) and high as the last index of the dictionary (2^k−1).
2. **Find the Midpoint**: In each step, calculate the midpoint of the current search space as mid = (low + high) / 2. Guess the word at index mid.
3. **Narrow Down**:
   - If the secret word is alphabetically after your guess, set low to mid + 1.
   - If the secret word is alphabetically before your guess, set high to mid - 1.
4. **Repeat**: Continue steps 2 and 3 until the secret word is found, which is when low exceeds high or you exactly match the guess.
5. **Success**: The algorithm guarantees finding the secret word when low exceeds **high** or you get an exact match, within at most k guesses.

Justification

- **Why it Works**: Each guess halves the search space. Starting with 2^k – 1 words, after the first guess, at most 2^(k−1) −1 words remain as candidates. After the second guess, at most 2^(k−2) − 1 words remain, and so on. This halving continues until the search space is reduced to 1 word, which must be the secret word.

- **Why at Most k Guesses**: The size of the dictionary is 2^k−1, and because each guess halves the search space, the maximum number of guesses needed to find the secret word is k. This follows from the property of binary search, where the search space is halved with each guess, leading to a logarithmic time complexity of O(log n with base 2) In this case, since n=2^k−1the logarithmic time complexity simplifies to O(k)

This algorithm is efficient and guarantees finding the secret word in at most k guesses, exploiting the binary division of the search space and the ordered nature of the dictionary.

c. To determine a recurrence relation for T(n), the maximum number of guesses required to identify a secret word from a dictionary with exactly n words, we reflect on the binary search used to efficiently find the word. Each guess effectively halves the search space, indicating the problem can be divided into two subproblems, each with half the original number of words. However, since we're looking for the maximum number of guesses, and each guess reduces the search space, we don't technically divide into two subproblems as in traditional binary search but rather continue with one subproblem of half the size. Thus, the recurrence relation for T(n) with n>1is:

T(n)=T(n/2)+1

And for the base case, when n=1, we have:

T(1)=0

Explanation of the Recurrence Relation

- The term T(n/2) represents the maximum number of guesses required after the first guess, which halves the search space.
- The +1 accounts for the initial guess made before the search space is halved.

This recurrence relation is true because, at each step of the search process, we make one guess (accounted for by the +1) and then continue the search with half of the remaining words (n/2), reflecting the halving process of the binary search.

Applying the Master Theorem

For the recurrence relation T(n)=T(n/2)+1with the base case T(1)=0, we apply the Master Theorem to determine the asymptotic behavior of T(n). The Master Theorem is used for recurrences of the form T(n)=aT(n/b)+f(n).

For our recurrence relation:

- a=1 because we continue with only one half of the search space each time,
- b=2 because we're dividing n by 2 in each recursive call,
- f(n)=1reflects the constant work done at each step in making a guess.

This fits directly into the case where f(n)= theta (1), which is equivalent to theta(n^(log a base b)) where log b base a =log 1 base 2= 0. Therefore, f(n) matches theta(n^0) = theta (1) and according to the Master Theorem, this implies that:

T(n)=theta (log n)

This shows that the maximum number of guesses required to correctly identify a secret word from a dictionary with exactly n words grows logarithmically with the size of the dictionary, and reflects the efficiency of the binary search approach in minimizing the number of guesses needed.

d. We can calculate the expected number of guesses we would need to correctly identify the secret word from a dictionary of 267,751 words. Given that every guess costs $1, we can then assess if this expected number of guesses would be less than or equal to the $15 provided to us on playing.

Calculating the Expected Number of Guesses

Given a dictionary of 267,751 words, and assuming each word is equally likely to be chosen, the most efficient guessing strategy is binary search, as discussed previously. The binary search method would identify any word in the dictionary in log N base 2 guesses, where N is the number of words in the dictionary.

Given N=267,751, we calculate:

log2(267,751)base 2 ≈ 18.03

Rounding up, since you can't make a fraction of a guess, it would take up to 19 guesses to correctly identify the secret word in the worst-case scenario.

Cost Analysis

For each guess costing $1, the total expenditure for up to 19 guesses would be:

19 * $1 = $19

Expected Outcome

Starting with $15 to play the game, if it takes up to 19 guesses, costing $19, then the expected financial outcome would result in a loss. The cost of playing exceeds the initial amount provided, leading to a shortfall of:

$19 - $15 = $4

Conclusion

Based on the corrected calculation for the expected number of guesses using the binary search strategy, and considering each word in the dictionary has an equal probability of being chosen, it's expected that participating in the "Guess My Word" game under these conditions would result in a loss. Specifically, the expenditure to identify the word, at $19, overshoots the initial $15 provided for playing the game, culminating in an anticipated loss of $4. This outcome suggests that, from a financial standpoint, agreeing to play the game would not be advisable.

Problem #5

a.

To perform Merge Sort on the array A = [6, 5, 8, 7, 3, 4, 2, 1], we'll follow the divide-and-conquer approach and illustrate each step visually.

Breakdown of Steps and Comparisons

- Initially, we have the array: **[6, 5, 8, 7, 3, 4, 2, 1]**

First Level of Division:

- Divide into **[6, 5, 8, 7]** and **[3, 4, 2, 1]**

Second Level of Division:

- **[6, 5, 8, 7]** divides into **[6, 5]** and **[8, 7]**
- **[3, 4, 2, 1]** divides into **[3, 4]** and **[2, 1]**

Third Level of Division:

- All sub-arrays are now single elements: **[6]**, **[5]**, **[8]**, **[7]**, **[3]**, **[4]**, **[2]**, **[1]**

Merging and Comparisons

First Level of Merging:

- Merge **[6]** and **[5]** requires 1 comparison.
- Merge **[8]** and **[7]** requires 1 comparison.
- Merge **[3]** and **[4]** requires 1 comparison.
- Merge **[2]** and **[1]** requires 1 comparison.

Second Level of Merging:

- Merge **[5, 6]** and **[7, 8]** requires 2 comparisons.
- Merge **[3, 4]** and **[1, 2]** requires 2 comparisons.

Final Level of Merging:

- Merging **[5, 6, 7, 8]** with **[1, 2, 3, 4]** requires 4 comparisons to get **[1, 2, 3, 4, 5, 6, 7, 8]**.

Total Comparisons

- **First Level Merging:** 4 comparisons (1 for each pair).
- **Second Level Merging:** 4 comparisons (2 for each merge).
- **Final Level Merging:** 4 comparisons.

The total comparisons are exactly 12. This corrected approach aligns with the merge sort's expected behaviour and described in the question.

Visual Illustration of the Merge Sort Process:

1. **Initial Array: [6, 5, 8, 7, 3, 4, 2, 1]**
2. **Divided into Sub-arrays:**
   - **[6, 5, 8, 7]** and **[3, 4, 2, 1]**
3. **Further Division:**
   - **[6, 5]**, **[8, 7]**, **[3, 4]**, **[2, 1]**
4. **Single Elements:**
   - **[6]**, **[5]**, **[8]**, **[7]**, **[3]**, **[4]**, **[2]**, **[1]**
5. **First Level of Merging:**
   - **[5, 6]**, **[7, 8]**, **[3, 4]**, **[1, 2]**
6. **Second Level of Merging:**
   - **[5, 6, 7, 8]**, **[1, 2, 3, 4]**
7. **Final Merge to Sorted Array:**
   - **[1, 2, 3, 4, 5, 6, 7, 8]**

Given this process, the visual representation would emphasize the divisions, the merging steps, and mark where each of the 12 comparisons takes place.

b. The equation **M(n)=2M(n/2) + n/2** can be understood in the context of sorting an even-numbered array as follows:

- **Divide:** The array of n elements is divided into two equal parts, each with n/2 elements. Sorting these parts independently involves recursive sorting, which, according to the function M, would require M(n/2) comparisons for each half. Since there are two halves, this step contributes 2M(n/2) comparisons to the total.
- **Conquer:** The conquer phase involves solving the simpler problem of sorting the smaller arrays. This is effectively done in the recursive call and is encapsulated in the 2M(n/2) term.
- **Combine (Merge):** The final step is to merge these two sorted halves back together. In the optimal case of merging, each element of one half is compared to the elements of the other half until the merge is complete. For an even-split, the minimum number of comparisons required to complete this merging is n/2, since each element from one half may only need to be compared once with the elements from the other half before finding its correct position in the sorted array.

So, the total minimum number of comparisons needed to sort the array is the sum of the comparisons needed to sort the two halves plus the comparisons needed to merge these sorted halves, which is mathematically represented as **M(n)=2M(n/2) + n/2**

c. To prove that M(n)=n * log(n) base 2/2for n being a power of 2, we'll use the principle of mathematical induction.

Base Case

Let's start with the smallest power of 2 for which the sorting operation is non-trivial, which is n=2.

For n=2, M(2)=1

Let's verify if it satisfies M(n)= n * log(n) base 2/2

M(2) = 2log(2) base 2/2= 2×1/2 = 1

The base case holds true.

Assumption and Inductive Step

Inductive steps:

Assume formula is true for $n = 2^k$, where $k \in N$.
That is, $M(2^k) = 2^k \log_2(2^k)/2$.

We know for merge sort: $M(2^{k+1}) = 2M(2^k) + 2^k$

On substitution: $M(2^{k+1}) = 2 \cdot (2^k \log_2(2^k)/2) + 2^k$

$$M(2^{k+1}) = 2^k \log_2(2^k) + 2^k$$

$$\therefore 2^k \log_2(2^k) = k \cdot 2^k$$

$$\therefore M(2^{k+1}) = k \cdot 2^k + 2^k$$

$$M(2^{k+1}) = 2^k(k+1)$$

but $2^k = 2^{k+1}/2$ and $k+1 = \log_2(2^{k+1})$

$$M(2^{k+1}) = (2^{k+1}/2) \log_2(2^{k+1})$$

$$M(2^{u+1}) = 2^{u+1} \cdot \log_2(2^{u+1})/2$$

This proves that if formula holds for $n = 2^k$, it also holds for $n = 2^{k+1}$, completing the inductive step.

Conclusion

By mathematical induction, we have shown that M(n)=n*log2(n) base 2/2 for n being a power of 2

d. Given A has 8 elements, the number of comparisons required to sort A using Merge Sort can vary depending on the initial arrangement of the elements in A. The Merge Sort algorithm involves dividing the array into two halves, recursively sorting each half, and then merging the two halves. The total number of comparisons required depends on both the recursive sorting and the merging steps.

Analysis:

For an array of 8 elements:

- **Divide:** The array is divided into two halves, each containing 4 elements.
- **Recursive Sorting:** Each half (with 4 elements) is sorted. Based on previous discussions, sorting an array of 4 elements using Merge Sort requires 4+4+2=10comparisons in total for both halves, considering the optimal scenario where each merge operation requires the minimum number of comparisons, which is 2 for merging two pairs of 2 elements each (since M(4) = 4).
- **Merging the Two Halves:** The final merge combines two sorted arrays of 4 elements each. The number of comparisons required in the final merge step can vary. In the best-case scenario, if the smallest 4 elements are all in one half and the largest 4 elements are all in the other half, 4 comparisons are required since each element from the first half is compared exactly once with the smallest element of the second half before being placed in the sorted array. This results in a total of 10+4=14 comparisons, not 12.

Observations for 12 Comparisons:

For Merge Sort to require exactly 12 comparisons for an array of 8 elements, we need to find a scenario that fits this requirement. However, the analysis above suggests that under the typical operation of Merge Sort, 14 comparisons are more characteristic for an 8-element array sorted under optimal conditions (with respect to minimizing comparisons).

Conclusion:

The probability that exactly 12 comparisons are required by Merge Sort to sort a random permutation of the array A = [1,2,3,4,5,6,7,8] does not align with the standard merge sort process for an array of this size, which generally suggests a higher number of comparisons (e.g., 14under optimal comparison conditions).

it can be concluded that, under normal circumstances, such a scenario is not expected, making the probability effectively 0 within the conventional understanding and operation of Merge Sort.

Problem #6:

a.

Breadth-First Search (BFS) explores the graph level by level. It starts from the root node and explores all the neighbouring nodes. Then, it moves to the next level of nodes.

Depth-First Search (DFS), on the other hand, explores as far as possible along each branch before backtracking. This means that it goes deep into the graph first before exploring the breadth.

Order of Visitation using BFS:

1. We start at the root node **a**.
2. We then visit **a**'s children **b** and **c**.
3. Next, we visit the children of **b** (**d**, **e**) followed by the children of **c** (**f**, **g**), as **b** comes before **c** in the alphabet.
4. After that, we visit the children of **f** (**h**, **i**) and **g** (**j**, **k**).
5. Then, we visit the children of **h** (**l**, **m**), followed by the children of **j** (**n**, **o**).
6. Finally, we visit the child of **k** (**p**) and the child of **m** (**q**).

The BFS order would be: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q.

Order of Visitation using DFS:

1. We start at the root node **a**.
2. We go down the leftmost path first, so we visit **a**'s child **b**.
3. We then visit **b**'s child **d**, as it is the left child.
4. Since **d** has no children, we backtrack to **b** and visit **b**'s right child **e**.
5. As **e** has no children, we complete **b**'s branch and move to **a**'s right child **c**.
6. We then explore **c**'s left child **f** and its children **h** and **l**, followed by **m** and **q**.
7. Next, we backtrack from **h** to **f** and visit **f**'s right child **i**.
8. After completing **f**'s branch, we move to **c**'s right child **g** and similarly explore **j**, **n**, **o**, and then **k** and **p**.

The DFS order would be**: a, b, d, e, c, f, h, l, m, q, i, g, j, n, o, k, p.**

Now, we will proceed to determine the order in which the 17 vertices are reached using each algorithm.
The order in which the 17 vertices are reached using Breadth-First Search (BFS) is:

a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q.

For Depth-First Search (DFS), the order is:

a, b, d, e, c, f, h, l, m, q, i, g, j, n, o, k, p.

These orders represent the paths taken by BFS and DFS algorithms when traversing the tree with the given rules of always picking the vertex that appears earlier in the alphabet when faced with a choice.

b.

The process of walking through an undirected binary tree by crossing each edge exactly twice, once in each direction, is known as a "Eulerian traversal". This is possible because a tree is a special case of a graph where any two vertices are connected by exactly one path.

a simple algorithm to achieve this Eulerian traversal:

1. Start at the root of the tree or any node as a starting point since it's undirected.
2. Move to an adjacent, unvisited node.
3. On reaching a node with no unvisited adjacent nodes, backtrack to the previous node.
4. Repeat steps 2 and 3 until you return to the starting node.

This algorithm works because each time we move from one node to an adjacent node, we cross an edge. When we backtrack, we cross the same edge again in the opposite direction. Since a tree is acyclic and connected, we will be able to visit all nodes and thus cross all edges twice.

Why each edge is guaranteed to be crossed exactly twice:

- The first time an edge is crossed when moving from one node to an adjacent node.
- The second time is when backtracking occurs because there are no more unvisited adjacent nodes.

The running time of this algorithm:

- Each edge is crossed exactly twice, and there are $n-1$ edges in a tree with n vertices (since a tree is a connected graph with n vertices and $n-1$ edges).
- Thus, the total number of edge crossings is $2(n-1)$. The algorithm spends a constant amount of time on each edge crossing.
- Therefore, the running time of this algorithm is $O(2(n-1))$, which simplifies to $O(n)$, linear in the number of vertices.

c. The diameter of a tree is the length of the longest path between any two vertices in the tree. This path is also known as the longest path or the maximum distance between any two nodes.

To find the diameter of the tree, we typically do the following:

1. Pick a random node and perform a Depth-First Search (DFS) or Breadth-First Search (BFS) to find the node that is farthest from it. This is known as an "extremal" node.
2. Perform a new DFS/BFS starting from the extremal node found in step 1 to find the node that is farthest from it.
3. The distance between these two extremal nodes is the diameter of the tree.

In the tree provided, to verify that the diameter is 7, we can apply the above steps:

- If we start a DFS from one of the leaves, say **q**, we will find that the farthest node from **q** is **p** (or **o**, **n**, which are equidistant). The path from **q** to **p** is **q-m-h-f-c-g-k-p**.
- Counting the edges along this path, we have: **q** to **m** (1), **m** to **h** (2), **h** to **f** (3), **f** to **c** (4), **c** to **g** (5), **g** to **k** (6), **k** to **p** (7).

Since no other path in this tree will have more than 7 edges, we can conclude that the diameter of this tree is 7. This is the maximum distance between any two vertices in the tree.

d. To compute the diameter of an undirected binary tree T with n vertices, we can use the following algorithm, which is based on two depth-first searches (DFS):

Algorithm:

1. **First DFS to find the farthest node from an arbitrary start node:**
   - Start a DFS from an arbitrary node (usually the root if given, otherwise any node will do).
   - During the DFS, keep track of the node that is the farthest from the start node. This is done by maintaining a variable that records the maximum distance encountered so far and the corresponding node.
2. **Second DFS to find the diameter:**
   - Start a new DFS from the farthest node found in step 1.
   - Similarly, keep track of the node that is the farthest from this node and the distance to it.
   - The distance to this farthest node is the diameter of the tree.

Explanation of the Algorithm:

- The reason for starting the second DFS from the farthest node found in the first DFS is based on the property of trees that one endpoint of the diameter will be the farthest node from any given node.

- The first DFS ensures that we find one endpoint of the potential diameter, and the second DFS measures the actual diameter by finding the farthest node from that endpoint.

Correctness Guarantee:

- The algorithm guarantees the correct output because of the properties of the tree. A tree is a connected acyclic graph, which means there is exactly one simple path between any two nodes.
- By definition, the diameter is the longest path between any two nodes in a tree.
- The first DFS finds a node that is the farthest from a starting node, which must be at one end of the diameter because if there were a longer path, it would have been discovered during the DFS due to the tree's acyclic nature.
- The second DFS then finds the farthest node from this extremal node, which by the tree's properties, is the other end of the diameter.

Running Time:

- Each DFS has a running time of O(n), where n is the number of nodes in the tree.
- Since we are performing two DFS operations sequentially, the total running time is O(n)+O(n)=O(2n)
- Therefore, the running time of the algorithm is O(n), linear in the number of vertices of the tree.