

CS 146: Data Structures and Algorithms

Midterm Exam 1

Instructor: Maryam Khazaei

Fall 2025

Date: September 30th, 2025

Duration: 75 minutes

Total Points: 90

Midterm Exam 1

This exam consists of **6 sections**, each covering a distinct topic. The order of the problems is randomized, so the sequence of questions may differ between students. It is recommended that you spend approximately **12–13 minutes per section** to manage your time effectively. Read each question carefully, show your work clearly, and do your best. If any question is unclear or ambiguous, state your assumptions and they will be taken into account when grading.

- **True/False** questions [Score: 15]
- **Recursion tree analysis** [Score: 15]
- **Max/Min Heap** [Score: 15]
- **Asymptotic notation and Master Theorem** [Score: 15]
- **Sorting Pseudocode** [Score: 15]
- **Data Structures** [Score: 15]

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. Case 1: If

$$f(n) = O(n^{\log_b a - \varepsilon}) \quad \text{for some constant } \varepsilon > 0,$$

then

$$T(n) = \Theta(n^{\log_b a}).$$

2. Case 2: If

$$f(n) = \Theta(n^{\log_b a}),$$

then

$$T(n) = \Theta(n^{\log_b a} \log n).$$

3. Case 3: If

$$f(n) = \Omega(n^{\log_b a + \varepsilon}) \quad \text{for some constant } \varepsilon > 0,$$

and if

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

for some constant $c < 1$ and all sufficiently large n , then

$$T(n) = \Theta(f(n)).$$

Asymptotic Notation:

$$f(n) = O(g(n)) \iff \exists c > 0, n_0 : \forall n \geq n_0, f(n) \leq c g(n)$$

$$f(n) = \Omega(g(n)) \iff \exists c > 0, n_0 : \forall n \geq n_0, f(n) \geq c g(n)$$

$$f(n) = \Theta(g(n)) \iff \exists c_1, c_2 > 0, n_0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Geometric Series:

$$S = \sum_{i=0}^n ar^i = a \frac{1 - r^{n+1}}{1 - r} \quad \text{for } r \neq 1$$

Special cases:

$$\text{If } |r| < 1 \text{ and } n \rightarrow \infty, \quad S = \frac{a}{1 - r}$$

$$\text{If } r = 1, \quad S = a(n + 1)$$

True or False

Select the correct response. No explanation is needed.

1. True / False : $\frac{n^3}{50} = O(n^2 + 10n^3)$

Solution: True, $\frac{n^3}{50}$ is $O(n^3)$, and $10n^3$ dominates n^2 .

2. True / False : Consider sorting an array of n elements using a comparison-based sorting algorithm and a decision tree. The longest path from the root to any leaf node in the tree must have $\Omega(n \log n)$ edges.

Solution: True, the longest path length from root to leaf in the decision tree for comparison sorting is at least $\Omega(n \log n)$.

3. True / False : The worst-case running time of Randomized Quicksort is $O(n^2)$.

Solution: True, the worst-case running time of Randomized Quicksort happens when the pivot selections consistently lead to highly unbalanced partitions. While this is not common, this scenario defines the worst-case behavior. In contrast, the expected running time is:

$$O(n \log n)$$

4. True / False : Decision trees can help visualize all possible comparisons and outcomes of a sorting algorithm on a fixed input size.

Solution: True

5. True / False : In order to prove $f(n) = O(n^3)$, we must find constants c and n_0 to satisfy an inequality involving both $f(n)$ and n^3 .

Solution: True, to prove that:

$$f(n) = O(n^3)$$

we must find constants $c > 0$ and $n_0 \geq 0$ such that:

$$f(n) \leq c \cdot n^3 \quad \text{for all } n \geq n_0$$

This satisfies the formal definition of Big-O notation. This is the formal definition of Big-O notation, which describes an upper bound on the growth rate of a function.

6. True / False : A decision tree representing a comparison-based sorting algorithm on n elements must have exactly $n!$ leaf nodes.

Solution: False, each leaf corresponds to a permutation of the input. There are $n!$ permutations, so there must be at least $n!$ leaves. The number of leaves is at least $n!$, but not necessarily exactly $n!$. If the algorithm has redundant or inefficient logic, it might have more than $n!$ leaf nodes, due to unnecessary comparisons or duplicated paths.

7. True / False : A binary tree with height h can have at most 2^h leaf nodes.

Solution: True, a binary tree with height h can have at most 2^h leaves.

8. True / False : As long as elements are placed into the correct buckets based on their values, Bucket Sort will produce a sorted output even without sorting the elements inside each bucket.

Solution: False, Bucket Sort requires that each bucket be individually sorted. Without sorting within buckets, the final concatenated output may not be correctly sorted.

9. True / False : In Bucket Sort, choosing too many buckets can lead to inefficient sorting due to increased overhead and many empty buckets.

Solution: True, choosing too many buckets can cause many buckets to be empty, which wastes space. Additionally, data may be unevenly distributed, which causes some buckets to contain many elements while others have few or none. To avoid these issues, the number of buckets should be chosen based on the input size and distribution.

10. True / False : In asymptotic notation, the function $f(n) = 0$ is in $\Theta(n)$.

Solution: False, $f(n) = 0$ is in $O(1)$ because the function is constant, not $\Theta(n)$ which describes tight bounds.

11. True / False : It takes at least $\Omega(n \log n)$ time for Mergesort (in the textbook) to sort any input of n elements.

Solution: True, Mergesort has a worst-case and average-case time complexity of $\Theta(n \log n)$. Thus, its lower bound is $\Omega(n \log n)$ for all inputs of size n , it means that it takes at least $\Omega(n \log n)$ time to sort any input of n elements.

12. True / False : The expression $\Omega(n)$ for insertion sort indicates that the algorithm's best case involves quadratic time.

Solution: False, $\Omega(n)$ gives a best-case lower bound, not quadratic.

13. True / False : Bucket Sort sorts each bucket using Counting Sort by default.

Solution: False, Bucket Sort does not use Counting Sort by default to sort the contents of each bucket. Instead, after distributing the elements into buckets, it uses a simple sorting algorithm such as Insertion Sort to sort the items within each bucket because the buckets usually contain only a few elements, and Insertion Sort works well on small datasets. It can also use other sorting algorithm such as quick sort, and counting sort

14. True / False : Counting Sort is a comparison-based sorting algorithm with worst-case time complexity $O(n \log n)$.

Solution: False, Counting Sort is not comparison-based and runs in $O(n + k)$ time. n = number of elements to sort, and k = range of input values.

15. True / False : In the definition of $\Omega(g(n))$, we require $f(n) \leq c \cdot g(n)$ for large n .

Solution: False, definition of Ω requires $f(n) \geq c \cdot g(n)$ for large n .

16. True / False : $n^2 + 100n = O(2^n)$

Solution: True, 2^n dominates the given polynomial.

17. True / False : For sufficiently large n , $\log n = O(n^\epsilon)$ for any $\epsilon > 0$.

Solution: True, $\log n = O(n^\epsilon)$ for any $\epsilon > 0$. You can check this intuitive by plugging in any small positive ϵ and checking the growth rates.

18. True / False : The expected running time of the randomized quicksort algorithm (that chooses the pivot randomly) is $O(n \log n)$ for all inputs of n distinct elements.

Solution: True, randomized quicksort has expected $O(n \log n)$ for all inputs.

19. True / False : The for-loop in insertion sort contributes to $O(n)$ of the runtime in the worst case.

Solution: False, in worst case, insertion sort does $O(n^2)$ work, not $O(n)$.

20. True / False : $n^{50} = \Omega(2^{n/2})$

Solution: False, polynomial n^{50} grows slower than exponential $2^{n/2}$.

21. True / False : Asymptotic notation only applies to time complexity, not space complexity.

Solution: False, asymptotic notation applies to both time and space.

22. True / False : The worst-case running time of randomized quicksort is always $O(n \log n)$, regardless of pivot choices.

Solution: False, randomized quicksort's worst-case is $O(n^2)$, not always $O(n \log n)$.

23. True / False : Counting Sort is a stable sorting algorithm because it processes the input array from right to left when building the output array.

Solution: True, When building the output array, Counting Sort uses a count array to determine the position of each element. To preserve the relative order of equal elements, it processes the original array from the end (right) toward the beginning (left).

24. True / False : In the best-case scenario, insertion sort performs zero comparisons in the inner loop.

Solution: False, even in best case, inner loop does at least one comparison.

25. True / False : $\log \log n = O(\log n)$

Solution: True, $\log \log n$ grows much slower than $\log n$ as n becomes large. Therefore, $\log \log n$ is asymptotically upper bounded by $\log n$.

26. True / False : In insertion sort, each element in the sorted portion may be shifted at most once during one outer loop iteration.

Solution: False, a value might be shifted multiple times in one iteration.

27. True / False : In the RAM model, operations like addition and copy are assumed to take variable time depending on the input size.

Solution: False, RAM model assumes each operation takes constant time.

28. True / False : Every comparison-based sorting algorithm has $\Omega(n \log n)$ worst-case time for inputs of size n .

Solution: False, lower bound applies to worst case only, not all inputs.

29. True / False : If $f(n) = O(n)$, then $f(n) \neq \Omega(n^2)$, for any function $f(n)$.

Solution: True, $n^2 > n$ for $n > 1$.

30. True / False : The lower bound on the number of comparisons any comparison-based sorting algorithm must make in the worst case is $\Omega(n \log n)$.

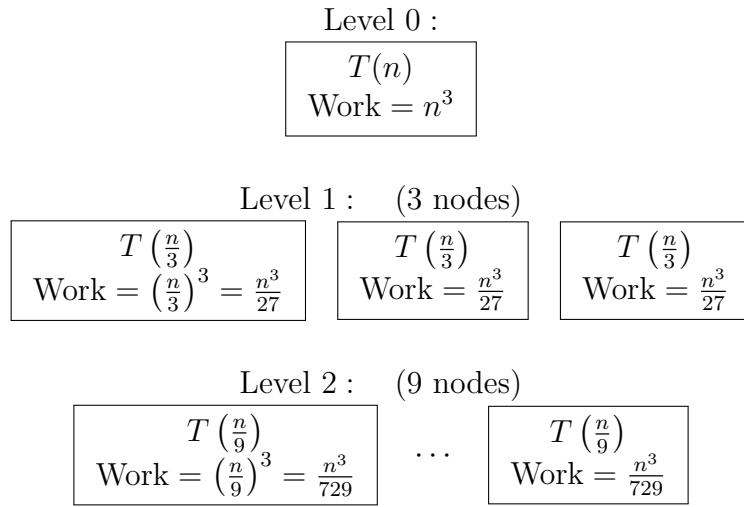
Solution: True, lower bound from decision tree argument is $\Omega(n \log n)$.

Recurrence Relation and Recursion Tree

Consider the recurrence relation:

$$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n^3)$$

Recursion Tree Visualization



Answer all the following questions. For each answer, clearly label it with the corresponding question number

1. What is the depth of the recursion tree in terms of n ?
2. How many subproblems are there at level d ?
3. What is the size of each subproblem at depth d ?
4. What is the amount of work done by each subproblem at depth d (not including any children/parents)?
5. Calculate the total amount of work done at level d .
6. Using the sum of the work at all levels, what is the asymptotic complexity $T(n)$?
7. In one or two sentences, describe a problem which may have this recurrence relation,

Solutions - Recurrence Relation and Recursion Tree

We observe that additional information can be obtained at level d :

$$\begin{cases} \text{Number of nodes} = 3^d \\ \text{Work per node} = \left(\frac{n}{3^d}\right)^3 = \frac{n^3}{3^{3d}} \\ \text{Total work at level } d = 3^d \times \frac{n^3}{3^{3d}} = \frac{n^3}{3^{2d}} = \frac{n^3}{9^d} \end{cases}$$

1. The recursion stops when the subproblem size is 1, if we solve the following equation, the depth of the tree is $\boxed{\log_3 n}$:

$$\frac{n}{3^d} = 1 \implies 3^d = n \implies d = \log_3 n$$

Solutions as

$$\log_2 n$$

are acceptable since

$$\log_b n = a \log_c n$$

where a, b, c are constants.

2. How many subproblems are there at level d ?

At each level, the number of subproblems multiplies by 3. Thus,

$$\boxed{3^d}$$

3. What is the size of each subproblem at depth d ?

Each subproblem size is divided by 3 at each level, so at depth d :

$$\boxed{\frac{n}{3^d}}$$

4. What is the amount of work done by each subproblem at depth d (not including any children/parents)?

The work done at a node of size s is $\Theta(s^3)$, so:

- Total work at level 1:

$$3 \cdot \left(\frac{n^3}{27}\right) = \frac{n^3}{9}$$

- **Total work at level 2:**

$$9 \cdot \left(\frac{n^3}{729} \right) = \frac{n^3}{81}$$

- **In general, total work at level d :**

$$3^d \cdot \frac{n^3}{3^{3d}} = \frac{n^3}{3^{2d}} = n^3 \cdot 3^{-2d}$$

5. Calculate the total amount of work done at level d .

Total work at level d is:

$$(\# \text{ subproblems}) \times (\text{work per subproblem}) = 3^d \times \frac{n^3}{3^{3d}} = n^3 \times 3^{d-3d} = n^3 \times 3^{-2d}$$

So,

$$\boxed{\Theta(n^3 \cdot 3^{-2d})}$$

6. Using the sum of the work at all levels, what is the asymptotic complexity $T(n)$?

Sum over all levels from $d = 0$ to $d = \log_3 n$:

$$T(n) = \sum_{d=0}^{\log_3 n} \Theta(n^3 \cdot 3^{-2d}) = n^3 \sum_{d=0}^{\log_3 n} 3^{-2d}$$

The summation is a geometric series with ratio $r = \frac{1}{3^2} = \frac{1}{9} < 1$:

$$\sum_{d=0}^m r^d = \frac{1 - r^{m+1}}{1 - r}$$

So,

$$T(n) = n^3 \cdot \frac{1 - 3^{-2(\log_3 n + 1)}}{1 - \frac{1}{9}} = n^3 \cdot \frac{1 - 3^{-2\log_3 n - 2}}{1 - \frac{1}{9}}$$

Simplify the exponent:

$$3^{-2\log_3 n} = (3^{\log_3 n})^{-2} = n^{-2}$$

Therefore,

$$T(n) = n^3 \cdot \frac{1 - \frac{1}{9}n^{-2}}{\frac{8}{9}} = n^3 \cdot \frac{1 - \frac{1}{9n^2}}{\frac{8}{9}} = \frac{9}{8}n^3 \left(1 - \frac{1}{9n^2} \right)$$

As $n \rightarrow \infty$, the term $\frac{1}{9n^2}$ tends to 0, so:

$$\boxed{T(n) = \Theta(n^3)}$$

7. In one or two sentences, describe a problem which may have this recurrence relation.

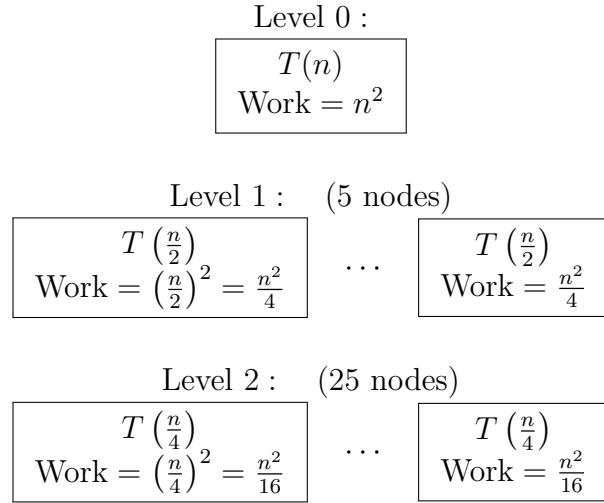
This recurrence

$$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n^3)$$

follows a divide-and-conquer algorithm that splits a problem of size n into 3 equal parts, then it solves each part recursively, and performs the running time of $\Theta(n^3)$ at each step.

Recurrence Relation and Recursion Tree

Consider the following Recursion Tree Visualization



Answer all the following questions. For each answer, clearly label it with the corresponding question number

1. What is the depth of the recursion tree in terms of n ?
2. How many subproblems are there at level d ?
3. What is the size of each subproblem at depth d ?
4. What is the amount of work done by each subproblem at depth d (not including any children/parents)?
5. Calculate the total amount of work done at level d .
6. Using the sum of the work at all levels, what is the asymptotic complexity $T(n)$?
7. In one or two sentences, describe a problem which may have this recurrence relation,

Solutions - Recurrence Relation and Recursion Tree

We observe that additional information can be obtained at level d :

$$\begin{cases} \text{Number of nodes} = 5^d \\ \text{Work per node} = \left(\frac{n}{2^d}\right)^2 = \frac{n^2}{4^d} \\ \text{Total work at level } d = 5^d \cdot \frac{n^2}{4^d} = \frac{n^2 \cdot 5^d}{4^d} \end{cases}$$

1. The recursion ends when the subproblem size becomes 1:

$$\frac{n}{2^d} = 1 \Rightarrow 2^d = n \Rightarrow d = \log_2 n$$

2. Number of subproblems at level d is:

$$5^d$$

3. Size of each subproblem at level d is:

$$\frac{n}{2^d}$$

4. Work per subproblem:

$$\left(\frac{n}{2^d}\right)^2 = \frac{n^2}{4^d}$$

5. Total work at level d :

$$5^d \cdot \frac{n^2}{4^d} = n^2 \cdot \left(\frac{5}{4}\right)^d$$

6. Total work across all levels:

$$T(n) = \sum_{d=0}^{\log_2 n} n^2 \cdot \left(\frac{5}{4}\right)^d$$

We apply the geometric series formula step-by-step to the original expression:

Given:

$$T(n) = \sum_{d=0}^{\log_2 n} n^2 \cdot \left(\frac{5}{4}\right)^d$$

$a = n^2$, $r = \frac{5}{4}$, and Upper limit: $d = \log_2 n$, so the number of terms is $\log_2 n + 1$

We apply the Geometric Sum Formula:

$$T(n) = n^2 \cdot \sum_{d=0}^{\log_2 n} \left(\frac{5}{4}\right)^d = n^2 \cdot \frac{1 - \left(\frac{5}{4}\right)^{\log_2 n + 1}}{1 - \frac{5}{4}}$$

Simplify the Denominator:

$$1 - \frac{5}{4} = -\frac{1}{4}$$

So:

$$T(n) = n^2 \cdot \frac{1 - \left(\frac{5}{4}\right)^{\log_2 n+1}}{-\frac{1}{4}} = -4n^2 \left(1 - \left(\frac{5}{4}\right)^{\log_2 n+1}\right)$$

Distribute the negative:

$$T(n) = 4n^2 \left(\left(\frac{5}{4}\right)^{\log_2 n+1} - 1\right)$$

We simplify $\left(\frac{5}{4}\right)^{\log_2 n+1}$ using exponent rules:

$$\left(\frac{5}{4}\right)^{\log_2 n+1} = \left(\frac{5}{4}\right) \cdot \left(\frac{5}{4}\right)^{\log_2 n} = \frac{5}{4} \cdot n^{\log_2\left(\frac{5}{4}\right)}$$

(using the identity $a^{\log_b n} = n^{\log_b a}$)

Final Expression will be as following:

$$T(n) = 4n^2 \left(\frac{5}{4} \cdot n^{\log_2\left(\frac{5}{4}\right)} - 1\right)$$

which is equal to (final answer):

$$T(n) = 5n^{2+\log_2\left(\frac{5}{4}\right)} - 4n^2$$

And for asymptotic analysis:

$$T(n) = \Theta\left(n^{2+\log_2\left(\frac{5}{4}\right)}\right)$$

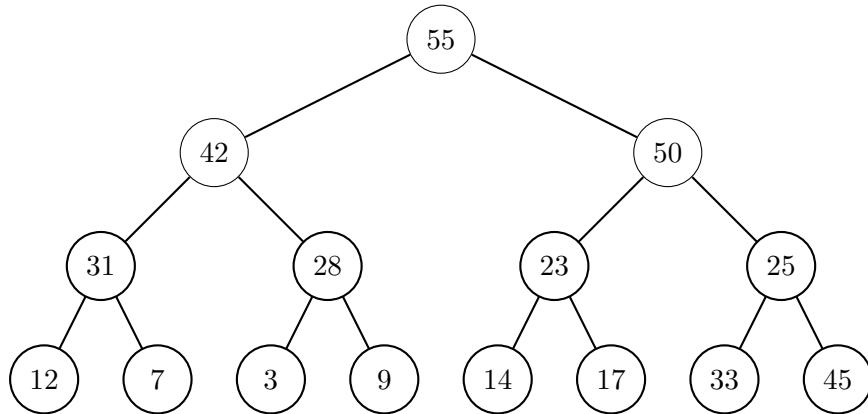
7. The recurrence relation:

$$T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n^2)$$

which represents a divide and-conquer algorithm that divides the problem into 2 subproblems of size $\frac{n}{2}$, and where the cost (run time) to divide and combine is $\Theta(n^2)$.

Max Heap

Below is a binary **max-heap**. Use the visualization to answer the following questions. Use the structure below to answer the following questions.



Questions

1. How many leaf nodes are in the heap?
2. What is the height/depth of the heap?
3. Write the array representation of the heap.
4. If this is a valid max-heap, explain why. If not, explain why not.
5. Assume the current max-heap is valid. If the root node (55) was replaced with node (12), how many swaps would be needed to restore the max-heap property using heapify-down? Explain your solution briefly.
6. Which node (or nodes) would violate the max-heap property if it (or they) replaced the root? Explain your solution briefly.
7. If a new element (60) is inserted into the heap, where will it first be placed, and what is its final position after heapify-up?

Solutions – Max Heap

1. How many leaf nodes are in the heap?

There are 8 leaf nodes in the heap: 12, 7, 3, 9, 14, 17, 33, and 45.

2. What is the height/depth of the heap?

The height of the heap is 3 (starting from 0 at the root). Recall from the lecture that the height of a tree is the number of edge from the root to the most distant leaf node.

3. Write the array representation of the heap.

The array representation is:

[55, 42, 50, 31, 28, 23, 25, 12, 7, 3, 9, 14, 17, 33, 45]

4. If this is a valid max-heap, explain why. If not, explain why not.

This is not a valid max-heap because the node with value 25 has left child and right child with values 33 and 45 respectively, which are greater than 25. In a max-heap, every parent must be greater than or equal to its children.

5. Assume the current max-heap is valid. If the root node (55) was replaced with node (12), how many swaps would be needed to restore the max-heap property using heapify-down? Explain your solution briefly.

Starting with 12 at the root: swap 12 with 50 (largest child), then swap 12 with 25, and finally swap 12 with 45. Thus, the total swaps is 3.

6. Which node (or nodes) would violate the max-heap property if it (or they) replaced the root? Explain your solution briefly.

Any node with value less than 50 violates the max-heap property if they will be replaced at the root, since the root must be at least as large as both children (42 and 50). Therefore, nodes such as 42, 31, 28, etc., will violate the max-heap property.

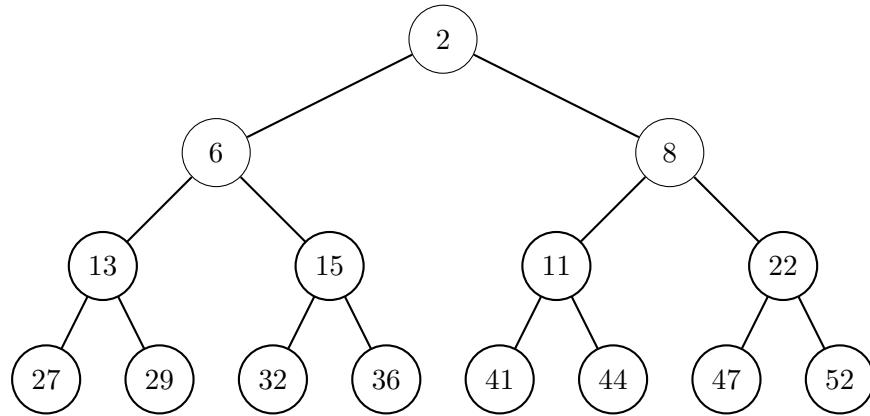
7. If a new element (60) is inserted into the heap, where will it first be placed, and what is its final position after heapify-up?

We initially, 60 is inserted at the next available position (as the left child of node 12). Heapify-up swaps will be as following: swap 60 with 12, then, swap 60 with 31, then, swap 60 with 42, and finally, swap 60 with 55. So, the final position is the root node.

Total swaps = 4

Min Heap

Below is a binary **min-heap**. Use the visualization to answer the following questions. Use the structure above to answer the following questions.



Questions

1. How many leaf nodes are in the heap?
2. What is the height/depth of the heap?
3. Write the array representation of the heap.
4. If this is a valid min-heap, explain why. If not, explain why not.
5. Assume the current min-heap is valid. If the root node (2) was replaced with node (27), how many swaps would be needed to restore the max-heap property using heapify-up? Explain your solution briefly.
6. Which node (or nodes) would violate the min-heap property if it (or they) replaced the root? Explain your solution briefly.
7. If a new element (1) is inserted into the heap, where will it first be placed, and what is its final position after heapify-down?

Solutions – Min Heap

1. How many leaf nodes are in the heap?

There are 8 leaf nodes in the heap: 27, 29, 32, 36, 41, 44, 47, 52 (recall that the leaf nodes are nodes with no children.)

2. What is the height/depth of the heap?

The height of the heap is 3 (starting from 0 at the root). Recall from the lecture that the height of a tree is the number of edge from the root to the most distant leaf node.

3. Write the array representation of the heap.

The array representation is:

[2, 6, 8, 13, 15, 11, 22, 27, 29, 32, 36, 41, 44, 47, 52]

4. If this is a valid min-heap, explain why. If not, explain why not.

This is a **valid min-heap** because every parent node is less than or equal to its children. For example: starting from the root, 2 is less than 6 and 8, then the left child of root 6 is less than 13 and 15, then the right child of root 8 is less than 11 and 22. We can continue this pattern to check all the nodes. Therefore, all parent-child relationships satisfy the min-heap property.

5. Assume the current min-heap is valid. If the root node (2) was replaced with node (27), how many swaps would be needed to restore the max-heap property using heapify-up? Explain your solution briefly.

Clearly max-heap property is a typo since the question title and every parts of the problem are related to min-heap.

Replace 2 with 27, and 27 with 2. Now. you need three swaps between "2 and "13, 6, and 27". Then there will be two more swaps between 27 and 6, and 27 and 13. So the total swaps are 5.

Note for students: If your answer was not completely correct, I still awarded partial credit.

6. Which node (or nodes) would violate the min-heap property if it (or they) replaced the root? Explain your solution briefly.

Any node greater than the children of root (6 and 8) will violate the min-heap property. So, any node with value larger than 6.

7. If a new element (1) is inserted into the heap, where will it first be placed, and what is its final position after heapify-down?

Initially, 1 is placed at the next available position in level order: index 15 (considering that index starts from zero), as the left child of 27. Compare 1 with its parent (27) and 1 less than 27. So we swap 1 and 27. Now compare 1 with 13. 1 is less than 13. So we swap 1 with 13. Now compare 1 with 6. 1 is less than 6. So we swap 1 with 6. Now compare 1 with 2. 1 is less than 2. So we swap 1 with 2. The final position is at the root and the total swaps is 4.

Asymptotic Notation and Master Theorem

1. Prove the following statement using the formal definition of asymptotic complexity:

$$n^2 \log n + 5n\sqrt{n} = O(n^2 \log n)$$

2. Solve the following recurrence using the Master Theorem. Clearly state the case used and justify all assumptions:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Solutions - Asymptotic Notation and Master Theorem

1. Prove the following statement using the formal definition of asymptotic complexity:

$$n^2 \log n + 5n\sqrt{n} = O(n^2 \log n)$$

Solution:

We prove this Using the formal definition of Big-O:

A function $f(n) = O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that:

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$

Let:

$$f(n) = n^2 \log n + 5n\sqrt{n}, \quad g(n) = n^2 \log n$$

Note that:

$$5n\sqrt{n} = 5n^{3/2}$$

Comparing $n^{3/2}$ with $n^2 \log n$ we find that:

$$\frac{n^{3/2}}{n^2 \log n} = \frac{1}{n^{1/2} \log n} \rightarrow 0 \quad \text{as } n \rightarrow \infty$$

Therefore, we need only find a single n_0 value where the inequality holds. We arbitrarily choose $n_0 = 16$ and calculate:

$$5(16)\sqrt{16} \leq 16^2 \log_2 16$$

$$5 \leq 4 * 4$$

$$5 \leq 20$$

Using the above observations we can substitute:

$$f(n) = n^2 \log n + 5n^{3/2} = O(n^2 \log n + n^2 \log n) = O(2n^2 \log n)$$

Therefore we can choose $c = 2$ and $n_0 = 16$.

Note for students: You must justify why n_0 , n , and c are the correct choices. Some points will be deducted if you do not provide a justification.

2. Solve the recurrence using the Master Theorem:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Solution:

This recurrence is of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

$$a = 8, \quad b = 2, \quad f(n) = \Theta(n^2), \quad \text{let } \varepsilon = 1$$

So this will be related to Case 1 of the Master Theorem, since the assumption of this case holds by substituting a, b, and epsilon and computing the log:

Case 1: If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

Therefore:

$$T(n) = \Theta(n^3)$$

Asymptotic Notation and Master Theorem

1. Prove the following statement using the formal definition of asymptotic complexity:
(hint: you can choose any constant log base which is convenient)

$$n^3 = \Omega\left(\frac{n^3}{\log n}\right)$$

2. Solve the following recurrence using the Master Theorem. Clearly state the case used and justify all assumptions:

$$T(n) = 4T(n/2) + \Theta(n)$$

Solutions - Asymptotic Notation and Master Theorem

1. Prove that

$$n^3 = \Omega\left(\frac{n^3}{\log n}\right).$$

Solution:

Recall the formal definition of Ω notation:

$$f(n) = \Omega(g(n)) \iff \exists c > 0, n_0 > 0 \text{ such that } \forall n \geq n_0, f(n) \geq c \cdot g(n).$$

Let

$$f(n) = n^3, \quad g(n) = \frac{n^3}{\log n}.$$

We want to find constants $c > 0$ and n_0 such that for all $n \geq n_0$,

$$n^3 \geq c \cdot \frac{n^3}{\log n}.$$

Divide both sides by n^3 :

$$1 \geq \frac{c}{\log n}$$

Thus,

$$\log n \geq c.$$

Since $\log n$ grows toward infinity as $n \rightarrow \infty$, we can pick any constant $c > 0$, and there will exist a valid n_0 . Let us arbitrarily choose $c = 1$, then:

$$\log_2 n_0 = 1$$

$$n_0 = 2$$

Therefore, we can choose $c = 1$ and $n_0 = 2$.

Note for students: You must justify why n_0 , n , and c are the correct choices. Some points will be deducted if you do not provide a justification.

2. Solve the recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n).$$

Solution:

The recurrence has the form

$$T(n) = a T\left(\frac{n}{b}\right) + f(n),$$

where

$$a = 4, \quad b = 2, \quad f(n) = \Theta(n), \quad \text{let } \varepsilon = 1$$

So this will be related to Case 1 of the Master Theorem, since the assumption of this case holds by substituting a, b, and epsilon and computing the log:

Case 1: If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then:

$$T(n) = \Theta(n^{\log_b a})$$

Note,

$$\log_b a = \log_2 4 = 2.$$

Therefore,

$$T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2).$$

Sorting Pseudocode

A student attempts to implement Merge Sort but is unsure how to write the `Merge` function. Instead of using the standard merge step, the student decides to sort the entire subarray using an alternative sorting algorithm after the recursive calls. The modified algorithm is shown below.

```
Merge-Sort(A, p, r)
1. if p == r return
2. q = floor((p + r) / 2)
3. Merge-Sort(A, p, q)
4. Merge-Sort(A, q + 1, r)
5. Heap-Sort(A, p, r)
```

Let $T(n)$ be the worst-case running time of this algorithm, where $n = r - p + 1$.

1. Write the recurrence relation for $T(n)$. Solve the recurrence for the time complexity of this approach using the Master Theorem.
2. In your own words, why does this student's approach have a longer running time than the correct Merge Sort?
3. Write the pseudocode for the correct `Merge` function.

Solutions - Sorting Pseudocode

Part 1

Each call to Merge-Sort makes two recursive calls on halves of the array, each of size approximately $n/2$, and then calls Heap-Sort on the full subarray of size n .

Heap Sort runs in worst-case time $\Theta(n \log n)$, so the recurrence is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n \log n) & \text{if } n > 1 \end{cases}$$

Master theorem cases do not apply on this problem.

To determine whether the Master Theorem applies, we compare $f(n) = \Theta(n \log n)$ with $n^{\log_b a} = n$:

- **Case 1** requires $f(n) = n \log n = O(n^{\log_b a - \epsilon}) = O(n^{\log_2 2 - \epsilon})$, which does not hold since $f(n)$ grows faster than $O(n^{\log_2 2 - 1})$
- **Case 2** requires $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$, but $f(n) = \Theta(n \log n)$ is asymptotically larger than n , so this case also does not apply.
- **Case 3** requires $f(n) = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{\log_2 2 + \epsilon})$, which is true. However, the regularity condition:

$$af(n/b) \leq cf(n) \text{ for some } c < 1$$

is only true for $c = 1$. Therefore, Case 3 does not apply either.

Since none of the cases apply, the Master Theorem cannot be used to solve this recurrence.

Note for students: No points will be deducted if you do not solve this part using the recursion tree method. However, you must clearly state that the Master Theorem does not apply to this recurrence and briefly explain why.

Solving with Recursion Tree

We now use the recursion tree method.

At each level i :

- There are 2^i subproblems, each of size $n/2^i$
- Cost per subproblem: $\frac{n}{2^i} \cdot \log\left(\frac{n}{2^i}\right)$
- Total cost at level i :

$$2^i \cdot \frac{n}{2^i} \cdot \log\left(\frac{n}{2^i}\right) = n \log\left(\frac{n}{2^i}\right) = n(\log n - i)$$

Total number of levels is $\log n$, so total cost is:

$$T(n) = \sum_{i=0}^{\log n-1} n(\log n - i) = n \sum_{i=0}^{\log n-1} (\log n - i)$$

Let $j = \log n - i$, then:

$$\sum_{i=0}^{\log n-1} (\log n - i) = \sum_{j=1}^{\log n} j = \frac{\log n(\log n + 1)}{2}$$

So:

$$T(n) = n \cdot \frac{\log n(\log n + 1)}{2} = \Theta(n \log^2 n)$$

Final Answer:

$T(n) = \Theta(n \log^2 n)$

Part 2

Merge Sort uses $\Theta(n)$ running time for the merge step at each level of recursion, and its total running time $\Theta(n \log n)$ (Please see the lecture slides for more explanations). However, this student's version uses Heap Sort on the entire subarray of size n , which takes $\Theta(n \log n)$ at each level, instead of just $\Theta(n)$. Since there are $\log n$ levels, the total work is $\Theta(n \log^2 n)$, which is asymptotically worse than the original $\Theta(n \log n)$.

Part 3

```

Merge(A, p, q, r)
1. n1 = q - p + 1
2. n2 = r - q
3. Let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4. for i = 1 to n1
5.     L[i] = A[p + i - 1]
6. for j = 1 to n2
7.     R[j] = A[q + j]
8. L[n1 + 1] = infinity
9. R[n2 + 1] = infinity
10. i = 1, j = 1
11. for k = p to r
12.     if L[i] <= R[j]

```

```
13.          A[ k ] = L[ i ]
14.          i = i + 1
15.      else
16.          A[ k ] = R[ j ]
17.          j = j + 1
```

Sorting Pseudocode

A student attempts to implement Merge Sort but is unsure how to write the `Merge` function. Instead of using the standard merge step, the student decides to sort the entire subarray using an alternative sorting algorithm after the recursive calls. The modified algorithm is shown below.

```
Merge-Sort(A, p, r)
1. if p == r return
2. q = floor((p + r) / 2)
3. Merge-Sort(A, p, q)
4. Merge-Sort(A, q + 1, r)
5. Counting-Sort(A, p, r)
```

Let $T(n)$ be the worst-case running time of this algorithm, where $n = r - p + 1$.

1. Write the recurrence relation for $T(n)$. Solve the recurrence for the time complexity of this approach using the Master Theorem.
2. In your own words, why does this student's approach have a longer running time than the correct Merge Sort?
3. Write the pseudocode for the correct `Merge` function.

Solutions - Sorting Pseudocode

1. The recurrence for $T(n)$ is:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

for some constant c . After splitting the array into two halves, the algorithm recursively sorts each half, each of size $n/2$, then it uses Counting-Sort on the entire subarray of size n .

Since Counting-Sort runs in $O(n + k)$ time, where k is the range of input values, and by assuming $k = O(n)$ (for simplicity and also for small k), Counting-Sort running time is $O(n)$.

Solving by Master Theorem:

The recurrence is of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where

$$a = 2, \quad b = 2, \quad f(n) = cn = \Theta(n).$$

Computing the following and then comparing $f(n)$ with $n^{\log_b a}$:

$$\log_b a = \log_2 2 = 1.$$

and

$$f(n) = \Theta(n), \quad n^{\log_b a} = n^1 = n.$$

Since $f(n) = n = \Theta(n^{\log_b a})$, this problem is related to case 2 of the Master Theorem, which gives:

$$T(n) = \Theta(n \log n).$$

2. In regular Merge Sort, the merge step takes $O(n)$ time and efficiently combines two sorted halves by comparing elements one by one.

The student's version replaces that with Counting Sort on the entire subarray. Counting Sort takes $O(n)$ time, but it does not consider that two halves are already sorted. So, instead of merging, the algorithm resorts the entire subarray at each level. This makes it less efficient in practice even though the overall time is still $O(n \log n)$, it takes longer than standard Merge Sort because it does more unnecessary sorting. Note that Merging two sorted halves using merge sort is faster because it uses the fact that the data is already half way sorted.

3. Merge(A, p, q, r)

1. $n1 = q - p + 1$

2. $n2 = r - q$

3. Let $L[1..n1 + 1]$ and $R[1..n2 + 1]$ be new arrays

4. for $i = 1$ to $n1$

```
5.      L[ i ] = A[ p + i - 1 ]
6.  for j = 1 to n2
7.      R[ j ] = A[ q + j ]
8.  L[ n1 + 1 ] = infinity
9.  R[ n2 + 1 ] = infinity
10. i = 1, j = 1
11. for k = p to r
12.     if L[ i ] <= R[ j ]
13.         A[ k ] = L[ i ]
14.         i = i + 1
15.     else
16.         A[ k ] = R[ j ]
17.         j = j + 1
```

Data Structures

Given two arrays A and B , both of length n , write pseudocode for a function that returns `true` if and only if A and B contain exactly the same elements (including duplicates), and `false` otherwise.

Constraints:

- You cannot use hash tables or hashing-based data structures.
- Your algorithm must run in worst-case time $O(n \log n)$.

Solution - Data Structures

To see if two arrays A and B (both of length n) have the same elements with the same counts, sort them and compare each position one by one.

Pseudocode:

```
CompareArrays(A, B):
1. if length(A) != length(B):
2.     return false
3. Merge-Sort(A)
4. Merge-Sort(B)
5. for i = 0 to length(A) - 1:
6.     if A[i] != B[i]:
7.         return false
8. return true
```

Time Complexity: Sorting array A takes $O(n \log n)$ time, and sorting array B also takes $O(n \log n)$ time. After sorting, comparing the two arrays element by element requires $O(n)$ time. Therefore, the total running time of the algorithm is $O(n \log n)$.

Space Complexity: The sorting step requires extra space proportional to the size of the arrays, that is, $O(n)$ space. The comparison step uses only a small, fixed amount of memory, so its space cost is $O(1)$. Overall, the algorithm uses $O(n)$ additional space.

Data Structures

Given two arrays X and Y , each of length n , write pseudocode for a function that returns `true` if and only if X is a superset of Y (all elements in Y also exist in X , but not necessarily vice versa), and `false` otherwise. Note that both arrays may contain duplicates.

Constraints:

- Do not use hash tables or any hashing-based data structure.
- Your algorithm must run in worst-case time $O(n \log n)$.

Solutions - Data Structures

We want to determine if array X is a superset of array Y , meaning every element in Y appears in X , including duplicates. The arrays may contain duplicates, so counts matter.

Approach: Since hashing is not allowed and we need $O(n \log n)$ worst-case time, a sorting-based method works well. We sort both arrays X and Y . Then, iterate through Y , checking whether each element appears in X with at least the same frequency. To do this efficiently, use two pointers: one for X and one for Y .

Pseudocode:

```
IsSuperset(X, Y):
1. Merge-Sort(X)
2. Merge-Sort(Y)
3. i = 0
4. j = 0
5. while j < length(Y):
6.     while i < length(X) and X[i] < Y[j]:
7.         i = i + 1 // If a match is found, move i forward.
8.     if i == length(X) or X[i] != Y[j]:
9.         return false
10.    i = i + 1
11.    j = j + 1
12. return true
```

Explanation: i , and j are pointer for X and Y respectively. We check each element of Y one by one and try to find a match for it in X . Since both arrays are sorted, we can move through them efficiently. If any element in Y does not appear in X , we return false. If we reach the end of Y after matching each element, it means every element was found in X , so we return true.

Time Complexity: We sort both X and Y , which takes $O(n \log n)$ time for each. After sorting, we go through both arrays once to check for matches, which takes linear time. So in total, the algorithm runs in $O(n \log n)$ time.

Space Complexity: The sorting step needs extra space proportional to the size of the arrays, so it takes $O(n)$ (Recall how the merge function in Merge Sort defines two new arrays for the left and right subarrays.). The comparison part only needs a few variables, so it uses constant space. Overall, the extra space used is $O(n)$.