Exam 1 Notes

Time Complexity

Time complexity in algorithms is a measure that gives us an estimation of the time an algorithm takes to run as a function of the length of the input. It is expressed using Big O notation, which provides an upper bound on the growth rate of the runtime of an algorithm. Time complexity is important because it helps us predict the scalability of an algorithm and understand how it will perform as we work with larger datasets or more complex problems.

Big $\mathcal{O}$ Notation

Big $\mathcal{O}$ Notation is used to describe the upper bound of the algorithm's runtime complexity, indicating the worst-case scenario. Common types of runtimes are:

- $\mathcal{O}(1)$ **- Constant Time**: Time to complete is the same regardless of input size.
- $\mathcal{O}(\log(n))$ **- Logarithmic Time**: Time to complete increases logarithmically as input size increases.
- $\mathcal{O}(n)$ **- Linear Time**: Time to complete scales linearly with the input size.
- $\mathcal{O}(n\log(n))$ **- Linearithmic Time**: Time to complete is a combination of linear and logarithmic growth rates.
- $\mathcal{O}(n^2)$ **- Quadratic Time**: Time to complete scales with the square of the input size.
- $\mathcal{O}(2^n)$ **- Exponential Time**: Time to complete doubles with each additional input unit.

Big $\mathcal{O}$, Big Omega Ω, and Big Theta Θ represent different bounds of runtime complexity:

- **Big $\mathcal{O}$**: The upper bound, or worst-case complexity.
- **Big Omega Ω**: The lower bound, or best-case complexity.
- **Big Theta Θ**: An algorithm is Θ if it is both $\mathcal{O}$ and Ω, indicating a tight bound where the upper and lower bounds are the same.

Identifying Leading Terms

For a given algorithmic complexity, there is a recipe for identifying what the 'leading term' of an algorithm is. This helps us identify how fast an algorithm may grow as time goes on. The recipe for doing this is:

1. **Identify the Leading Term**: For large values of n the term with the highest exponent in n will have the most significant impact on the growth rate of the function. Constant factors are irrelevant in Big $\mathcal{O}$ notation.
2. **Simplify Logarithmic Expressions**: Convert all logarithms to the same base if possible, and remember that constants in front of logarithms (like 2 in $\log_2(n)$) do not change the complexity class. Use the change of base formula if needed.
3. **Compare Growth Rates**: Know the order of growth rates: $\log(n) < n < n\log(n) < n^k < a^n < n!$. This helps in quickly identifying which terms dominate as n grows large.
4. **Ignore Lower Order Terms and Coefficients**: When determining the Big $\mathcal{O}$ class, you can ignore any constants and any terms that grow more slowly than the leading term. For example, in $n^2 + 100n$, the $100n$ term is irrelevant for large n, and the function is $\mathcal{O}(n^2)$.
5. **Be Mindful of Exponentials**: Recognize that any exponential function a^n (where $a > 1$) will grow faster than any polynomial, and thus does not belong to $\mathcal{O}(n^k)$ for any constant k.
6. **Special Cases**: Be aware of functions that may look complex but simplify to a known growth rate, such as polynomial functions with non-integer exponents. Assess whether the polynomial growth dominates the logarithmic or constant factors.

Asymptotes

In the context of algorithms, the concept of asymptotes is related to the idea of asymptotic analysis, which is concerned with the behavior of algorithms as the size of the input grows very large. Here's a concise summary of how the concept relates to algorithms:

- **Asymptotic Behavior**: Asymptotic analysis looks at the limit of an algorithm's performance (time or space required) as the input size approaches infinity. It helps in understanding the efficiency of an algorithm in the worst case (usually).
- **Asymptotic Upper Bound (Big $\mathcal{O}$)**: This is like a 'ceiling' for the growth of an algorithm's running time. It means that the algorithm's running time will not exceed a certain boundary as the input size grows indefinitely.
- **Asymptotic Lower Bound (Big Ω)**: Analogous to a 'floor', it gives a guarantee that the algorithm's running time will be at least as high as the bound for sufficiently large inputs.
- **Tight Bound (Big Θ)**: If an algorithm has a Big Theta bound, it means that both the upper and lower bounds are the same asymptotically. The algorithm's running time grows at the same rate as the function in the Big Theta notation.
- **Asymptotic Equality**: When we say an algorithm has a time complexity of, say, $\mathcal{O}(n^2)$, we mean that the running time increases at most quadratically as n approaches infinity. We're not concerned with the exact match but the trend as n becomes very large.

Master's Method

The Master Method provides a method to analyze the time complexity of recursive algorithms, especially those following the divide and conquer approach.

Form of Recurrence

The theorem applies to recurrences of the form:

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

where:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. $b > 1$.
- $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and combining the results of the subproblems.

The Three Cases

The solution to the recurrence, $T(n)$, can be categorized into three cases based on the comparison between $f(n)$ and $n^{\log_b(a)}$ (the critical part of the non-recursive work). The constant ϵ is equated to be $\epsilon = \log_b(a)$, c is often referred to as the exponent of the leading term in $f(n)$.

1. **Case 1 (Divide or Conquer dominates)**
   - If $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$, essentially $\epsilon > c$.
     - Then $T(n) = \Theta(n^\epsilon)$
2. **Case 2 (Balanced)**
   - If $f(n) = \Theta(n^{\log_b(a)} \cdot \log^k(n))$ for some constant $k \geq 0$, essentially $\epsilon = c$.
     - Then $T(n) = \Theta(n^\epsilon \log(n))$
3. **Case 3 (Work outside divide/conquer dominates)**
   - If $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ for some constant $\epsilon > 0$, and if $af\left(\frac{a}{b}\right) \leq kf(n)$ for some constant $k < 1$ and large enough n, essentially $\epsilon < c$.
     - Then $T(n) = \Theta(f(n))$

Core Tenants Of Master's Method

- **Applicability**
  - The Master Method is particularly useful for analyzing the time complexity of algorithms that break problems down into smaller subproblems, each of a fixed proportion of the size of the original.
  - Common examples include merge sort, quicksort, and binary search algorithms.
- **Limitations**
  - It does not apply to all types of recurrences.
  - It cannot be used when the recursive subproblems are of unequal size.

Recursion Trees

Recursion trees provide a visual and intuitive method for solving recurrence relations in recursive algorithms, especially those in divide and conquer algorithms.

Concept

- A recursion tree is a tree where each node represents the cost of a certain part of the algorithm.
- The root represents the initial problem; children of a node represent subproblems that the algorithm divides the problem into.

Building a Recursion Tree

The recipe for building a recursion tree can be summarized below:

1. **Root Node**: Represents the initial problem size, n.
2. **Child Nodes**: Each level of the tree corresponds to a recursive call. The children of a node represent the subproblems into which the problem is divided.
3. **Cost at Each Node**: Write the cost of the work done at each level (excluding the recursive calls) on the corresponding node.
4. **Depth of the Tree**: Corresponds to the number of recursive calls before reaching the base case.

Analyzing the Tree

We can analyze a recursion tree with the following steps:

1. **Calculate Cost at Each Level**: Sum the costs of all nodes at each level of the tree.
2. **Total Cost**: The total cost of the algorithm is the sum of the costs at each level of the tree.