Part 1: Basic Concepts

1. What are the three primary criteria we seek when designing an algorithm?

Answer:

The three primary criteria we seek when designing an algorithm are:

1) Correctness: The algorithm should consistently produce correct solution given a valid input.
2) Effectiveness: The algorithm should use minimal computational resources such as time and memory for solving the given problem.
3) Easy to Implement: The algorithm should be simple, clear and easy to understand so that a programmer is less likely to make any mistakes during implementation.

2. List the five basic data structure types and their average time and space complexities.

Answer:

The five basics data structures along with their average time and space complexity are tabulated as follows:

Data Structure	Access	Search	Insertion	Deletion	Space Complexity
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)
Stack	—	—	O(1) (Push)	O(1) (Pop)	O(n)
Queue	—	—	O(1) (Enqueue)	O(1) (Dequeue)	O(n)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

3. What is the specific objective of the Interval Scheduling Problem?

Answer:

The specific objective of the Interval Scheduling Problem is to find the maximum numbers of non-overlapping tasks from a given set of tasks such that, no two selected tasks overlap with each other.

For example,

Given tasks {A, B, C, D, E} where each task occupies a specific time interval between time units 1 to 5 as shown in the table below:

Tasks/Time Interval	1	2	3	4	5
A	🟩	🟩			
B		🟩	🟩		
C			🟩	🟩	🟩
D			🟩		
E					🟩

The goal of the problem is to find the maximum number of non-overlapping tasks that can be performed. In this case {A, D, E} is the solution for the given problem.

4. List three different types of greedy strategies that could be used for interval scheduling.

Answer:

The three different types of greedy strategies that could be used for interval scheduling are:

1) Earliest Start Time: In this strategy, we select the first task with earliest start time and then repeatedly select the consecutive task which starts earliest and doesn't overlap with the one we selected.
2) Earliest Finish Time: In this strategy, we select the first task with earliest finish time and then repeatedly select the consecutive tasks which finish earliest and doesn't overlap with the ones we selected.
3) Shortest Interval: In this strategy, we select the tasks with shortest intervals ensuring that they don't overlap with each other.

5. What is the fundamental difference between an algorithm and a heuristic?

Answer:

The fundamental difference between an algorithm and a heuristic is as follows:

Algorithm	**Heuristic**
It is a well-defined set of states that guarantees correct and optimal solution (if exists) for a given problem.	It is a problem-solving approach that aims to find a near-optimal solution quickly without guaranteeing correctness or optimality.
Example: Brute-Force is an algorithm that tries all possible combination to provide correct and optimal solution.	Example: Greedy Methods are heuristics that make locally optimal choices to get fast and near optimal solutions.

Part 2: Application and Analysis

1. Describe the Earliest Finish Time greedy algorithm steps.

Answer:

The steps for Earliest Finish Time greedy algorithm are as follows:

1) Sort the given jobs in the increasing order of their finish time.
2) Initialize selected tasks list.
3) Select the first job from sorted jobs.
4) For each job in remaining jobs:
    a. Select the job that finishes early and does not overlap with the last job of the selected jobs.
    b. Repeat until all jobs are considered.

Output: List of non-overlapping tasks.

2. Why is the Nearest Neighbor approach for the Traveling Salesman Problem considered a heuristic rather than a correct algorithm?

Answer:

The Nearest Neighbor approach for the Traveling Salesman Problem is considered a heuristic rather than a correct algorithm because it is a greedy method that makes locally optimal choices by always visiting the nearest unvisited city. Although it runs fast and produces a valid tour, it does not guarantee the shortest possible solution.

3. Explain the concept of a Loop Invariant and its relation to mathematical induction.

Answer:

A loop invariant is a condition or property that holds true before the loop starts, remains true before and after each iteration of the loop, and is still true when the loop terminates. Loop invariants are used to prove the correctness of algorithms, especially iterative ones.

The correctness of a loop invariant is proven using mathematical induction:

1) Initialization (Base Case): The invariant is true before the first iteration of the loop.
2) Maintenance (Inductive Step): If the invariant is true before an iteration, it remains true after the iteration.
3) Termination (Conclusion): When the loop ends, the invariant combined with the termination condition proves that the algorithm produces the correct result.

Loop invariants rely on the same structure as mathematical induction, to prove algorithm correctness.

4. What does it mean for a problem like the Traveling Salesman Problem (TSP) to be NP-hard?

Answer:

For a problem like the Traveling Salesman Problem, being NP-hard means that there is no known algorithm that can solve the Traveling Salesman Problem in a polynomial time. However, according to theory of computation, any other NP-hard problem can be reduced to the Travelling Salesman Problem in polynomial time.

This problem can be solved using exhaustive search (brute force) with time complexity of O(n!). However, as the value of n (number of cities) grows, it becomes computationally infeasible to use this algorithm.

5. How does modularity contribute to an algorithm's ease of implementation?

Answer:

Modularity is the process of breaking down an algorithm into smaller, independent and manageable components called modules. Each module performs a well-defined task independently. Hence, making an algorithm modular can help break down a complex problem into simple modules making the algorithm easier to understand, test and implement contributing to the algorithm's ease of implementation.

Part 3: Proof and Complexity

1. **Proof by Contradiction**: Summarize the argument used to prove that the greedy algorithm for interval scheduling cannot select fewer jobs than an optimal schedule.

Answer:

To prove that the greedy algorithm for interval scheduling cannot select fewer jobs than an optimal schedule we first assume that the greedy algorithm selects jobs fewer than the optimal schedule. Considering the earliest finish time approach, our algorithm selects the job that has the earliest finish time. Assuming that the optimal solution selects different first job, replacing the first job of the optimal solution with the one selected by the greedy approach does not change the total number of jobs in optimal solution since the job selected by the greedy approach finishes earlier than the first job in optimal solution. We can repeat this replacement for each subsequent jobs showing that the greedy schedule can be transformed into the optimal schedule without selecting the fewer jobs. This is the contradiction to our initial assumption that the greedy algorithms select job fewer than the optimal schedule.

2. Counterexample Challenge: Draw or describe a set of intervals where the Shortest Interval greedy strategy fails to provide a maximum-size subset of compatible jobs.

Answer:

Given tasks {A, B, C, D, E} where each task occupies a specific time interval between units 1 to 12 as shown in the table below:

Tasks/Time Interval	1	2	3	4	5	6	7	8	9	10	11	12
A	🟩	🟩	🟩	🟩								
B				🟩	🟩							
C					🟩	🟩	🟩	🟩				
D								🟩	🟩			
E									🟩	🟩	🟩	🟩

We can see here that using the Shortest Interval greedy strategy returns the set of selected jobs as {B, D}. However, for the given problem the optimal solution is clearly {A, C, E}. Hence, in this case the Shortest Interval greedy strategy fails to provide maximum-size subset of compatible jobs.

3. Exhaustive Search Complexity: Why is exhaustive search (trying n! permutations) impractical for a Robot Tour Optimization with 20 points? Provide the approximate number of permutations involved.

Answer:

Given a Robot Tour Optimization problem with 20 points, using exhaustive search is impractical for this problem because we will have to search through total 20! permutations which is approximately 2.432×10^{18} which takes a very long time.

Suppose we have a computer that can perform 10^{10} computations. Then the time taken by the algorithm to find the optimal path = $= \frac{2.432 * 10^{18}\ (computations)}{10^{10}\ (computations\ /\ second)} = 2.432\ *\ 10^{8}$ seconds = 7.70036981 years which is too long.

4. Algorithm Correctness: The slides state that "Failure to find a counterexample... does not mean the algorithm is correct." Explain why mathematical induction is preferred over trial-and-error for proving correctness.

Answer:

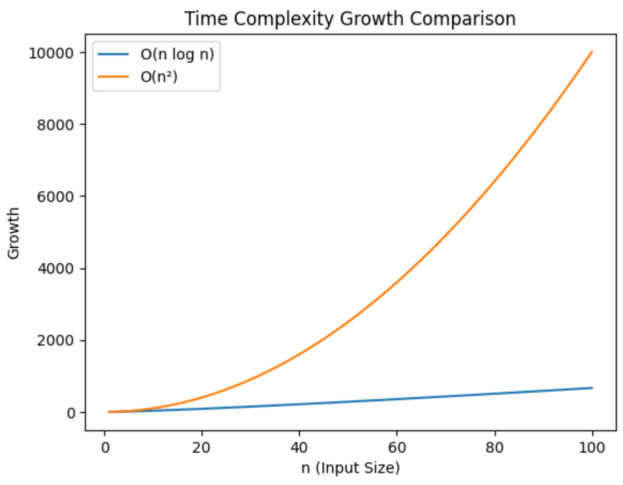
An algorithm usually has an infinitely countable set of inputs, so there may exist some input set S for which the algorithm fails, even if no counterexample has been found through testing. However, by using mathematical induction, we establish a formal proof showing that if the algorithm is correct for a base case and remains correct from one case to the next, then no such failing set S exists. This provides a basis for proving the algorithm's correctness.

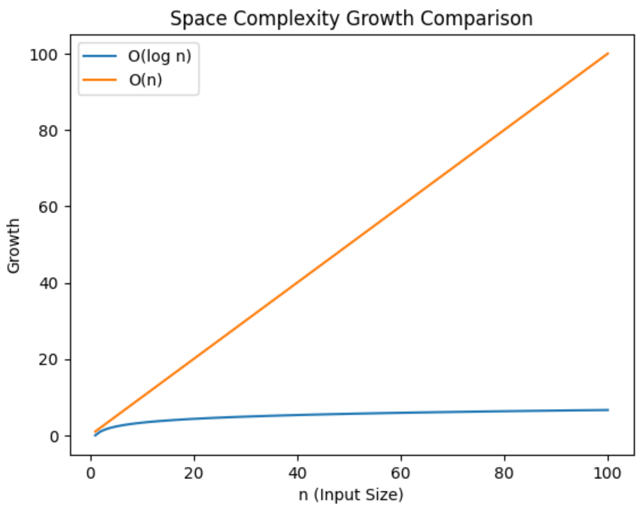
5. Complexity Trade-offs: Compare Quicksort and Mergesort based on their worst-case time complexity and space complexity.

Answer:

Comparison of Quicksort and Mergesort based on their worst-case time complexity and space complexity is as follows:

Algorithm	Worst-Case Time Complexity	Space Complexity
Quicksort	$O(n^2)$	O(log n)
Mergesort	O(n log n)	O(n)




Space Complexity Growth Comparison
O(log n)
O(n)
Growth
n (Input Size)
0
20
40
60
80
100

Empirical Study of Interval Scheduling Algorithms

1. Overview of Implementation

The coding task implements and empirically evaluates greedy heuristics and an exhaustive optimal solver for the Interval Scheduling Problem (maximizing non-overlapping intervals). Key components:

- **Dataset Generation**: Random intervals with start ~ Uniform(0, T), duration ~ Uniform(1, 100), where T = α * n * 100. α controls overlap: 0.1 (high), 1.0 (medium), 5.0 (low).
- **Algorithms**:
    - Greedy: EFT (sort by finish, select non-overlapping), EST (sort by start), SD (sort by duration). All use merge sort (O(n log n)).
    - Exhaustive: Enumerates 2^n subsets, validates non-overlap (O(n log n) per subset), finds max size. No pruning, leading to O(n 2^n log n) worst-case.
- **Experiments**: Quality ratios (small n=8-20), greedy runtime (n=2^10 to 2^19), exhaustive runtime (n=10-20). Averaged over trials.
- **Tools**: Python, NumPy, Matplotlib.

2. Solution Quality Analysis

Coding Question Addressed: Compare greedy strategies (EFT, EST, SD) against optimal for solution quality (ratio of selected intervals to optimal) across overlap regimes and n values.

Experiments: For n=8 to 20 (step 2), 20 trials per point, compute average ratio (greedy count / optimal count).

Key Findings

- **EFT**: Always achieves ratio 1.0 across all α and n, confirming optimality. This aligns with proof by contradiction: assuming an optimal solution has more jobs leads to a contradiction by showing the greedy algorithm's early finish leaves room for additional jobs.
- **EST and SD**: Heuristics that are not always optimal. Performance degrades with higher n and overlap:
    - High overlap (α=0.1): EST drops to ~0.7, SD to ~0.5. Dense conflicts punish poor choices (e.g., EST may select early-starting long jobs blocking others; SD prioritizes short jobs that block compatible longer ones).
    - Medium overlap (α=1.0): EST ~0.7-0.9, SD ~0.6. Moderate degradation.
    - Low overlap (α=5.0): All ~0.9-1.0, as sparsity reduces suboptimal impact.
- Observation: EFT's finish-time priority is crucial (proven optimal). Heuristics approximate well in sparse scenarios but fail in dense ones, highlighting why they're heuristics with no correctness guarantee.

Plot: Solution Quality: Greedy vs Optimal

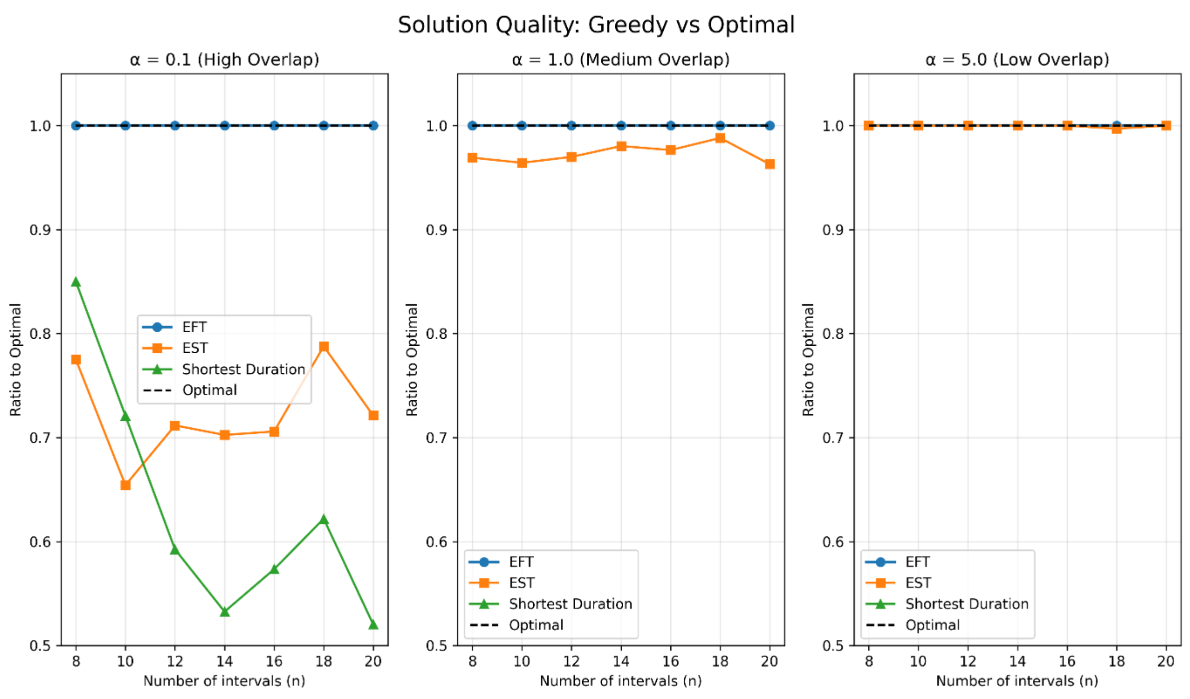


Table 1: Average Quality Ratios (Excerpt for n=20)

α (Overlap)	EFT Ratio	EST Ratio	SD Ratio	Optimal
0.1 (High)	1.0	0.721	0.520	1.0
1.0 (Medium)	1.0	0.963	0.258	1.0
5.0 (Low)	1.0	1.0	0.179	1.0

These results validate that SD has counterexamples (high-overlap case shows failure) and demonstrate the maintained non-overlap loop invariant in the greedy algorithm, proven via induction.

3. Runtime Complexity Validation

Coding Question Addressed: Empirically validate Big-O complexities for greedy (O(n log n)) and exhaustive (O(n 2^n)) across α, with normalization to confirm bounds.

3.1 Greedy Runtime (EFT as Representative)

Experiments: n=1024 to ~500k, 10 trials. Measures total time (sorting + selection).

- **Findings**:
  - Runtime scales as O(n log n): Log-log plot shows near-linear slope (slope ≈1 for n log n).
  - Normalized t(n)/(n $\log_2$ n): Stabilizes at ~2.2-2.9 × 10^{-7} s (constant, confirming bound). Initial fluctuations due to overhead at small n.

    - α Impact: Low overlap (α=5.0) slightly faster (fewer comparisons); high overlap slower but same complexity.
- Ties to Theory: Matches Mergesort's O(n log n) worst-case time complexity and O(n) space, compared to Quicksort's O(n^2) worst-case time but O(log n) space.

Plot: Greedy Runtime Validation

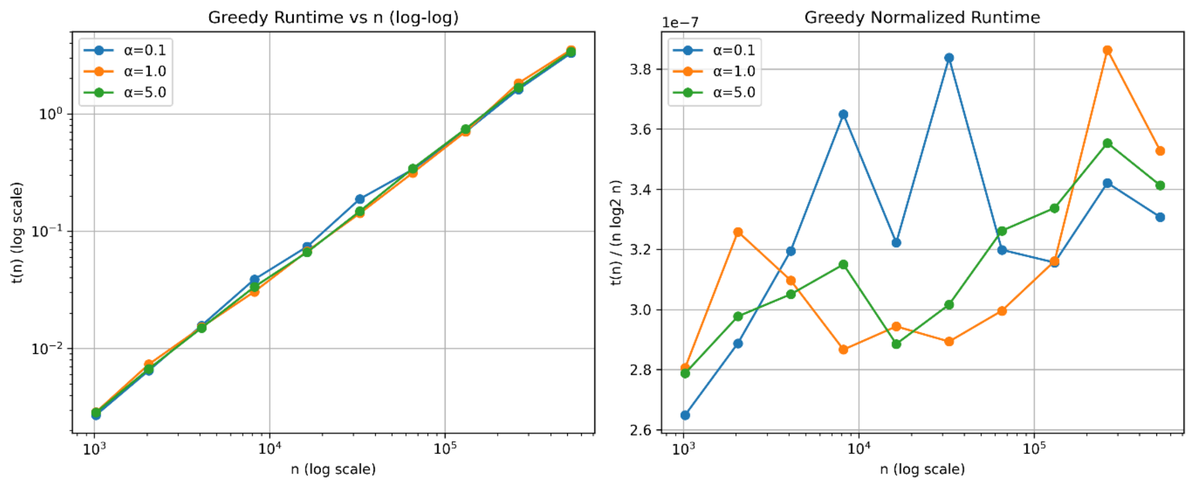


Table 2: Greedy Runtime Excerpt (α=1.0)

n	Time (ms)	Normalized (×10^{-7})
1024	2.87	2.81
16384	67.52	2.94
524288	3514.74	3.53

3.2 Exhaustive Runtime

Experiments: n=10-20, 5 trials.

- **Findings**:
    - Runtime O(n 2^n): vs n plot shows exponential growth (up to ~4s at n=20 for high overlap).
    - Normalized t(n)/(n 2^n): Decreases to ~1.6-2.4 × 10^{-7} s (trends toward constant but doesn't fully stabilize due to small n). High overlap (α=0.1) has lower constant (implicit early invalidations act as "pruning").
    - α Impact: High overlap faster in practice (more early conflicts skip computations), but worst-case still exponential.
- Ties to Theory: Impractical for n>20 (e.g., n=20: ~1 million subsets, compared to n! for TSP which is ~2.4 × 10^{18} for n=20, far worse). No explicit pruning, but backtracking could optimize average-case without changing worst-case. Exhaustive search is impractical at scale; induction proofs are preferred for correctness over trial-and-error.

Plot: Exhaustive Runtime Validation

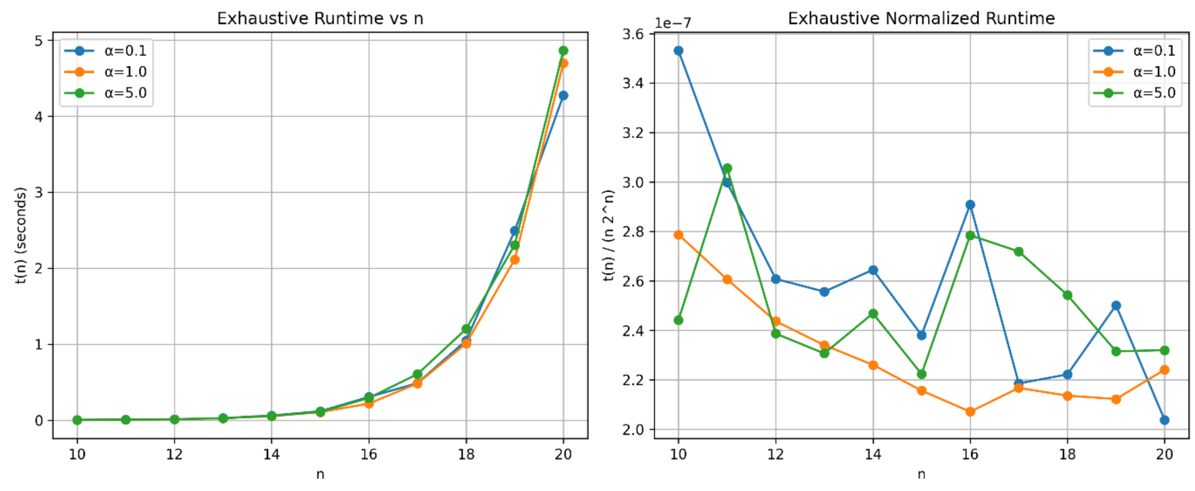


Table 3: Exhaustive Runtime Excerpt (α=0.1)

n	Time (s)	Normalized (×10^{-7})
10	0.0036	3.53
15	0.117	2.38
20	4.273	2.04

4. Discussion and Observations

- **Overlap Impact (α)**: High overlap amplifies heuristic failures (quality) but slightly improves exhaustive runtime (implicit pruning). Low overlap makes all strategies near-optimal.
- **Trade-offs**: Greedy efficient (scalable) but heuristics like EST/SD can be suboptimal, distinguishing them from proven algorithms. Exhaustive is optimal but unscalable (NP-hard problem similar to TSP).
- **Limitations**: Small n for exhaustive limits normalization; floating-point precision in intervals; no pruning in exhaustive.
- **Improvements**: Use dynamic programming for optimal (O(n log n)) in quality tests; integer intervals; more α/trials.
- **Relation to Theory**: Empirical results support theoretical proofs (EFT optimality via contradiction/induction). Failure of heuristics in counterexamples shows why trial-and-error is insufficient.

5. Conclusion

The implementation confirms EFT's optimality and O(n log n) efficiency, while heuristics approximate variably based on overlap. Exhaustive validates optimality but highlights exponential impracticality. This empirical study complements theory, demonstrating criteria like correctness, efficiency, and simplicity.

All plots saved in 'plots/' (quality_comparison.png, greedy_big_o.png, exhaustive_big_o.png). Code runs via python main.py.