

# Algorithm Design: Greedy and Divide-and-Conquer Approaches for Task Scheduling and Skyline Computation

Chittela Venkata Sai Tarun Reddy  
University of Florida

## Abstract

We present two fundamental algorithmic paradigms applied to practical computational problems. First, we develop a greedy algorithm for task scheduling with deadlines, achieving optimal  $O(n \log n)$  time complexity through the Earliest Deadline First (EDF) strategy. We prove correctness via exchange argument and validate performance experimentally. Second, we design a divide-and-conquer algorithm for the skyline problem—computing visible building silhouettes from overlapping rectangles—with  $O(n \log n)$  complexity verified through Master Theorem analysis. Both algorithms are validated against baseline approaches with comprehensive experimental evaluation demonstrating theoretical predictions. Our implementations achieve measured slopes of 1.03-1.05 in log-log runtime plots, confirming the  $n \log n$  complexity bounds.

## Keywords

Greedy algorithms, Divide-and-conquer, Task scheduling, Skyline problem, Algorithm analysis

## 1 Introduction

Algorithm design paradigms provide powerful frameworks for solving computational problems efficiently. Two fundamental paradigms—greedy algorithms and divide-and-conquer—offer complementary approaches: greedy algorithms make locally optimal choices at each step, while divide-and-conquer recursively breaks problems into smaller subproblems.

This paper presents two problems solved using these paradigms:

**Problem A (Greedy):** Task Scheduling with Deadlines—maximizing the number of tasks completed before their deadlines in a single-processor system.

**Problem B (Divide-and-Conquer):** Skyline Problem—computing the visible silhouette of overlapping building rectangles.

Both problems have significant real-world applications in operating systems, cloud computing, computer graphics, and geographic information systems. We provide complete solutions including formal abstractions, algorithms, complexity analysis, correctness proofs, and experimental validation.

## 2 Problem A: Task Scheduling with Deadlines

### 2.1 Problem Domain and Motivation

In modern computing environments, task scheduling is fundamental to system performance. Consider a cloud computing platform processing batch jobs with Service Level Agreement (SLA) deadlines. Each job requires a specific execution time and must complete before its deadline. The scheduler must maximize the number of jobs meeting their SLAs.

Other applications include:

- **Operating Systems:** Process scheduling with real-time constraints
- **Project Management:** Prioritizing tasks to meet project milestones
- **Manufacturing:** Sequencing production orders with delivery dates

### 2.2 Problem Abstraction

**Input:** A set of tasks  $T = \{t_1, t_2, \dots, t_n\}$  where each task  $t_i$  has:

- Duration  $d_i \in \mathbb{Z}^+$  (execution time units)
- Deadline  $D_i \in \mathbb{Z}^+$  (must finish by this time)

**Output:** A subset  $S \subseteq T$  and ordering  $\sigma$  such that:

- (1) Each task  $t_i \in S$  completes before deadline  $D_i$
- (2)  $|S|$  is maximized (maximum number of schedulable tasks)

### 2.3 Algorithm: Earliest Deadline First (EDF)

Our greedy strategy schedules tasks in order of increasing deadline:

---

#### Algorithm 1 Greedy Task Scheduling (EDF)

---

**Require:** Tasks  $T = \{t_1, \dots, t_n\}$  with durations and deadlines

**Ensure:** Maximum subset  $S$  of schedulable tasks

```
1: Sort  $T$  by deadline:  $t_1.D \leq t_2.D \leq \dots \leq t_n.D$ 
2:  $S \leftarrow \emptyset$ 
3:  $currentTime \leftarrow 0$ 
4: for  $i = 1$  to  $n$  do
5:   if  $currentTime + t_i.d \leq t_i.D$  then
6:      $S \leftarrow S \cup \{t_i\}$ 
7:      $currentTime \leftarrow currentTime + t_i.d$ 
8:   end if
9: end for
10: return  $S$ 
```

---

### 2.4 Complexity Analysis

**THEOREM 1.** *The EDF algorithm runs in  $O(n \log n)$  time and uses  $O(n)$  space.*

**PROOF. Time Complexity:**

- Line 1 (sorting):  $O(n \log n)$  using efficient comparison-based sort
- Lines 4-9 (greedy selection):  $O(n)$  single pass through tasks
- Total:  $O(n \log n) + O(n) = O(n \log n)$

**Space Complexity:**

- Storage for  $n$  tasks:  $O(n)$
- Selected subset  $S$ :  $O(n)$  worst case

- Auxiliary space for sorting:  $O(n)$  or  $O(\log n)$  depending on implementation
- Total:  $O(n)$

□

## 2.5 Correctness Proof

We prove optimality using the *exchange argument*, a standard technique for greedy algorithms.

**THEOREM 2 (EDF OPTIMALITY).** *The Earliest Deadline First algorithm produces an optimal solution—it schedules the maximum number of tasks.*

**PROOF.** We prove by contradiction using an exchange argument.

Assume for contradiction that there exists an optimal solution  $O$  that differs from the greedy solution  $G$  produced by EDF.

Let position  $i$  be the first position where  $O$  and  $G$  differ. At position  $i$ :

- $G$  schedules task  $g$  with deadline  $D_g$
- $O$  schedules task  $o$  with deadline  $D_o$
- By EDF ordering:  $D_g \leq D_o$  (since  $g$  comes first in sorted order)

**Exchange Operation:** Construct solution  $O'$  by swapping tasks  $g$  and  $o$  in  $O$ :

- Task  $g$  (earlier deadline) now comes at position  $i$
- Task  $o$  (later deadline) comes later

**Feasibility of  $O'$ :**

- (1) Before position  $i$ :  $O$  and  $G$  are identical, so all tasks meet deadlines
- (2) At position  $i$  in  $O'$ : Task  $g$  meets its deadline since:
  - In  $G$ , task  $g$  meets deadline with same completion time
  - $O'$  has identical schedule up to position  $i$
  - Therefore  $g$  still meets deadline  $D_g$  in  $O'$
- (3) After position  $i$ : Task  $o$  now completes later. Since  $D_o \geq D_g$  and  $o$  met its deadline in  $O$  at earlier time, it still meets  $D_o$  at later time in  $O'$
- (4) All other tasks: No change in relative ordering, so feasibility preserved

Therefore  $|O'| = |O|$  and  $O'$  is optimal. Repeating this exchange for all differing positions shows  $G = O'$ , contradicting our assumption that  $O \neq G$ .

Hence the EDF algorithm is optimal. □

## 2.6 Experimental Validation

We implemented the algorithm in Python 3.11 and conducted two sets of experiments:

**Correctness Verification:** We compared EDF against brute force enumeration (testing all permutations) for  $n = 8$  tasks over 30 trials. All trials confirmed optimality (Table 1).

**Performance Benchmarking:** We measured runtime for  $n \in \{200, 400, 800, 1600, 3200, 6400\}$  with 100 trials each (Figure 1). Log-log regression yielded slope = 1.03, confirming  $O(n \log n)$  complexity.

**Table 1: Scheduling Correctness Verification (n=8, 30 trials)**

Metric	Greedy (EDF)	Brute Force
Trials Matched	30/30 (100%)	-
Avg Tasks Scheduled	8.0	8.0
Execution Time	0.00001s	0.02s



**Figure 1: Greedy Task Scheduling Runtime.** Log-log plot shows measured runtime (blue) closely follows fitted  $O(n \log n)$  curve (red) with slope 1.03.

## 3 Problem B: Skyline Problem

### 3.1 Problem Domain and Motivation

The skyline problem arises in computer graphics, geographic information systems, and urban planning. Given multiple overlapping building rectangles, compute the visible silhouette as seen from a distance.

Applications include:

- **Computer Graphics:** Efficient rendering of 3D city scenes
- **GIS Systems:** Urban planning and visualization
- **Video Games:** Real-time skyline rendering
- **Architecture:** Analyzing building visibility and shadows

### 3.2 Problem Abstraction

**Input:** A set of buildings  $B = \{b_1, b_2, \dots, b_n\}$  where each building  $b_i$  is a rectangle defined by:

- Left edge  $L_i \in \mathbb{Z}$
- Right edge  $R_i \in \mathbb{Z}$  where  $R_i > L_i$
- Height  $H_i \in \mathbb{Z}^+$

**Output:** Skyline  $S$  as an ordered sequence of key points:

$$S = [(x_1, h_1), (x_2, h_2), \dots, (x_k, h_k)]$$

where the height changes from  $h_i$  to  $h_{i+1}$  at position  $x_{i+1}$ .

The skyline represents the upper envelope of all buildings when viewed from infinite distance.

### 3.3 Algorithm: Divide-and-Conquer Skyline

Our approach recursively divides buildings, computes skylines for each half, then merges:

---

#### Algorithm 2 Divide-and-Conquer Skyline

---

**Require:** Buildings  $B = \{b_1, \dots, b_n\}$   
**Ensure:** Skyline  $S$  as list of  $(x, h)$  key points

```

1: if  $|B| = 0$  then
2:   return []
3: end if
4: if  $|B| = 1$  then
5:   return  $[(b_1.L, b_1.H), (b_1.R, 0)]$ 
6: end if
7: mid  $\leftarrow \lfloor n/2 \rfloor$ 
8:  $S_{\text{left}} \leftarrow \text{Skyline}(B[1..mid])$ 
9:  $S_{\text{right}} \leftarrow \text{Skyline}(B[mid + 1..n])$ 
10: return Merge( $S_{\text{left}}, S_{\text{right}}$ )

```

---



---

#### Algorithm 3 Merge Two Skylines

---

**Require:** Skylines  $S_1, S_2$

**Ensure:** Merged skyline

```

1:  $S \leftarrow [], i \leftarrow 1, j \leftarrow 1, h_1 \leftarrow 0, h_2 \leftarrow 0$ 
2: while  $i \leq |S_1|$  and  $j \leq |S_2|$  do
3:   if  $S_1[i].x < S_2[j].x$  then
4:      $x \leftarrow S_1[i].x, h_1 \leftarrow S_1[i].h, i \leftarrow i + 1$ 
5:   else if  $S_2[j].x < S_1[i].x$  then
6:      $x \leftarrow S_2[j].x, h_2 \leftarrow S_2[j].h, j \leftarrow j + 1$ 
7:   else
8:      $x \leftarrow S_1[i].x, h_1 \leftarrow S_1[i].h, h_2 \leftarrow S_2[j].h$ 
9:      $i \leftarrow i + 1, j \leftarrow j + 1$ 
10:    same x-coordinate
11:   end if
12:    $h_{\max} \leftarrow \max(h_1, h_2)$ 
13:   if  $|S| = 0$  or  $S[\text{last}].h \neq h_{\max}$  then
14:     Append  $(x, h_{\max})$  to  $S$ 
15:   end if
16: end while
17: Append remaining points from  $S_1[i..]$  and  $S_2[j..]$  to  $S$ 
18: return  $S$ 

```

---

### 3.4 Complexity Analysis

**THEOREM 3.** *The divide-and-conquer skyline algorithm runs in  $O(n \log n)$  time and uses  $O(n)$  space.*

**PROOF. Time Complexity:**

Let  $T(n)$  be the time to compute a skyline for  $n$  buildings.

*Recurrence Relation:*

- Base case:  $T(1) = O(1)$  (lines 3-5)
- Recursive case:

- Two recursive calls:  $2T(n/2)$  (lines 7-8)
- Merge operation:  $O(|S_1| + |S_2|) = O(n)$  since total key points  $\leq 2n$

Therefore:  $T(n) = 2T(n/2) + O(n)$

*Master Theorem Application:*

Compare  $T(n) = 2T(n/2) + O(n)$  to standard form  $T(n) = aT(n/b) + f(n)$ :

- $a = 2, b = 2, f(n) = O(n)$
- $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
- Since  $f(n) = \Theta(n^{\log_b a})$ , we have Case 2
- By Master Theorem:  $T(n) = \Theta(n \log n)$

**Space Complexity:**

- Skyline storage:  $O(n)$  (at most  $2n$  key points)
- Recursion depth:  $O(\log n)$
- Total space:  $O(n) + O(\log n) = O(n)$

□

### 3.5 Correctness Proof

**THEOREM 4 (SKYLINE CORRECTNESS).** *The divide-and-conquer algorithm correctly computes the skyline.*

**PROOF.** We prove by strong induction on the number of buildings  $n$ .

**Base Case ( $n = 1$ ):** A single building produces skyline  $[(L, H), (R, 0)]$ , which correctly represents one rectangle.

**Inductive Hypothesis:** Assume the algorithm correctly computes skylines for all inputs with fewer than  $n$  buildings.

**Inductive Step ( $n$  buildings):** Divide buildings into left half  $B_L$  and right half  $B_R$  with  $|B_L|, |B_R| < n$ .

By the inductive hypothesis:

- $S_L$  correctly represents skyline of  $B_L$
- $S_R$  correctly represents skyline of  $B_R$

**Merge Correctness:** The merge operation constructs the combined skyline by maintaining:

- (1) Current heights  $h_1$  (from  $S_L$ ) and  $h_2$  (from  $S_R$ )
- (2) At each x-coordinate, combined height =  $\max(h_1, h_2)$
- (3) Key points added only when height changes

For any x-coordinate  $x$  in the final skyline:

- The height is  $\max(\text{height from } S_L, \text{height from } S_R)$
- This equals the maximum building height at  $x$  across all buildings
- This is precisely the definition of the combined skyline

Therefore, the merged skyline correctly represents all  $n$  buildings.

By strong induction, the algorithm is correct for all  $n \geq 1$ . □

### 3.6 Experimental Validation

**Correctness Verification:** We compared divide-and-conquer against a naive  $O(n^2)$  algorithm for  $n = 20$  buildings over 30 trials. All skylines matched exactly (Table 2).

**Performance Benchmarking:** We measured runtime for  $n \in \{100, 200, 400, 800, 1600, 3200\}$  with 50 trials each. Results confirm  $O(n \log n)$  complexity:

The comparison plot (Figure 3) dramatically illustrates the advantage of divide-and-conquer, with the gap widening as  $n$  increases.

**Table 2: Skyline Correctness Verification (n=20, 30 trials)**

Metric	D&C	Naive
Trials Matched	30/30 (100%)	-
Avg Skyline Points	18.2	18.2
Avg Execution Time	0.0002s	0.005s
Speedup	-	25x

**Figure 2: Divide-and-Conquer Skyline Runtime. Log-log plot shows measured runtime (blue) matches fitted  $O(n \log n)$  curve (red) with slope 1.05.**

## 4 Related Work

**Task Scheduling:** The EDF algorithm is a classic result in real-time systems [1]. Extensions include multi-processor scheduling and preemptive variants.

**Skyline Problem:** First formalized by Bentley [2], the divide-and-conquer approach has been extended to 3D and dynamic scenarios in computational geometry.

## 5 Conclusion

We presented two algorithmic paradigms applied to practical problems:

- (1) **Greedy (Task Scheduling):** The Earliest Deadline First algorithm achieves optimality through simple local choices, proven via exchange argument.
- (2) **Divide-and-Conquer (Skyline):** Recursive decomposition and merging yields efficient  $O(n \log n)$  solution, proven via Master Theorem.

Both algorithms were rigorously proven correct and experimentally validated. Measured slopes (1.03-1.05 in log-log plots) closely match theoretical  $O(n \log n)$  predictions, demonstrating the power

**Figure 3: Algorithm Comparison. D&C  $O(n \log n)$  (blue) significantly outperforms naive  $O(n^2)$  (red) as problem size grows.**

of these algorithmic paradigms for real-world computational problems.

**Future Work:** Extensions include multi-processor scheduling variants, 3D skyline problems, and dynamic/online versions where tasks or buildings arrive incrementally.

## References

- [1] C. L. Liu and J. W. Layland. *Scheduling algorithms for multiprogramming in a hard-real-time environment*. Journal of the ACM, 20(1):46–61, 1973.
- [2] J. L. Bentley. *Multidimensional divide-and-conquer*. Communications of the ACM, 23(4):214–229, 1980.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 3rd edition. MIT Press, 2009.
- [4] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson, 2006.

## A Implementation Code

The complete Python implementation (algorithm\_project.py) is provided below. The code includes:

- Both greedy and divide-and-conquer algorithms
- Baseline comparison algorithms (brute force, naive)
- Correctness verification tests
- Performance benchmarking with statistical analysis
- Automated CSV data export and plot generation

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Algorithm Design Project: Greedy & Divide-and-Conquer
=====
Author: Chittela Venkata Sai Tarun Reddy
Date: November 2024
Python: 3.11+
A) GREEDY: Task Scheduling with Deadlines
    - Domain: Operating systems, cloud computing
    - Algorithm: Earliest Deadline First (EDF)
    - Complexity: O(n log n) time, O(n) space
B) DIVIDE-AND-CONQUER: Skyline Problem
    - Domain: Computer graphics, GIS systems
    - Algorithm: Recursive skyline merging
    - Complexity: O(n log n) time, O(n) space
"""

[Full code provided in artifact - see algorithm_project.py]
```

*Note: The complete implementation code is included in the artifact file. Due to space constraints, we show only the header here. The full 600+ line implementation includes all algorithms, experiments, and plotting functionality.*

## B LLM Usage Disclosure

This project was completed with assistance from Large Language Models (LLMs) as permitted by the course guidelines. Below we document all LLM interactions:

### B.1 Tools Used

- **Claude 3.5 Sonnet** (Anthropic): Primary assistance for algorithm design, proof structuring, and LaTeX formatting
- **Usage**: Interactive problem-solving and document preparation

### B.2 Prompts and Interactions

#### Initial Prompt (Problem Selection):

*"I need to design two algorithms for my class project: one using greedy approach and one using divide-and-conquer. They must be real-world problems not from textbooks. Can you suggest suitable problems with different domains than battery charging and temperature detection?"*

**LLM Response Summary:** Suggested task scheduling with deadlines (greedy) and skyline problem (divide-and-conquer), with justification for their practical applications.

#### Algorithm Design Prompts:

- "Provide pseudocode for Earliest Deadline First scheduling algorithm"
- "Help me design the divide-and-conquer skyline merge function"

#### Proof Assistance Prompts:

- "How do I prove the greedy task scheduling algorithm is optimal? Guide me through an exchange argument."
- "Explain how to apply Master Theorem to the skyline recurrence  $T(n) = 2T(n/2) + O(n)$ "

#### LaTeX Formatting Prompts:

- "Format this proof as a LaTeX theorem environment with proper mathematical notation"
- "Create ACM conference style document with algorithm pseudocode blocks"

## B.3 Verification Process

All LLM-generated content was:

- (1) **Verified for correctness**: Mathematical proofs were checked step-by-step
- (2) **Tested experimentally**: Algorithms implemented and validated with comprehensive test suites
- (3) **Cross-referenced**: Claims verified against textbook sources (Cormen et al., Kleinberg & Tardos)
- (4) **Adapted and refined**: Original LLM suggestions were modified based on experimental results

**Student Contribution:** We take full responsibility for the correctness of all proofs, algorithms, and experimental results. The LLM served as a guide and formatting assistant, but all final validation and verification was performed manually.

## C Code Repository

The complete source code, including all implementations, experimental data, and generated plots, is available in the following GitHub repository:

<https://github.com/chittela2003/AOA-project1>

The repository contains:

- `algorithm_project.py`: Complete Python implementation of both algorithms
- Experimental results and CSV data files
- Generated plots (PNG format) used in this paper
- README with setup and execution instructions