# Greedy and Divide-and-Conquer: Case Studies in NewsItem Scheduling and 2D Range Reporting

Yingzhu Chen
y.chen6@ufl.edu
UFID: 83945095
University of Florida, USA

Forrest Yan Sun
yan.sun@ufl.edu
UFID: 47948015
University of Florida, USA

## ABSTRACT

This project studies two algorithmic paradigms: greedy algorithms and divide-and-conquer. We present two complete case studies: (1) a NewsItem scheduling problem, optimally solved by a greedy algorithm using Disjoint Set Union (DSU); and (2) a 2D orthogonal range reporting problem for Point of Interest (POI) lookups, solved using a divide-and-conquer k-d tree. For each problem, we provide the real-world identification, formal abstraction, algorithm design, complexity analysis, and correctness proof. We also implement both optimized algorithms and verify their theoretical running times against naive baselines through empirical experiments.

## KEYWORDS

Greedy algorithms, disjoint set union, scheduling, NewsItem, divide-and-conquer, orthogonal range reporting, k-d tree

## 1 INTRODUCTION

Many practical decision problems admit efficient, simple greedy solutions that are optimal under natural assumptions. On a university campus such as the University of Florida, hundreds of student activities, talks, workshops, and club events occur every day. Each activity (we model as a NewsItem) has a period during which it is relevant: after the event occurs, its timeliness and utility for attending drop sharply. Students have limited time and attention, and activities have different initial importance to a student depending on major, interests, or professional relevance.

We therefore study a NewsItem scheduling problem: the platform (student app) should recommend and schedule which campus activities to present or highlight in limited front-page slots so that students can be guided to attend events that maximize their overall benefit. Each NewsItem has an initial base value which depends on its topical relevance (for instance, how closely it matches a student's major), and a deadline (the last slot before it becomes irrelevant). The scheduler must select and order NewsItems under slot-capacity constraints to maximize total realized student utility.

For the divide-and-conquer half of the project we selected a different, practical problem from spatial databases: orthogonal 2D range reporting (report all restaurants or points of interest that lie inside an axis-aligned query rectangle) . This problem is central to applications like maps, where a naive linear scan is too slow. This problem admits divide-and-conquer style solutions. In Part II, we implement and analyze a k-d tree, a classic D&C data structure, to solve this problem and verify its performance empirically.

The report is organized into two main sections : Part I — Greedy for NewsItem scheduling, and Part II — Divide-and-Conquer for 2D range reporting. Under each main section we follow the course rubric: (1) real problem identification; (2) abstraction; (3) algorithm and code; (4) running-time analysis; (5) correctness/intuitive explanation and experimental plan.

## 2 PART I — GREEDY (NEWSITEM SCHEDULING)

### 2.1 Real problem identification

Practical problem: campus activity recommendation and scheduling for University of Florida students. Every day a large number of campus events (seminars, club meetings, career talks, workshops, social activities) are announced. Each event has a short window during which attending is possible and valuable; afterwards the event has little to no practical value. Students have limited attention and time; recommending a small number of events in prominent slots (e.g., home screen or daily digest) can significantly impact participation and student outcomes.

Each NewsItem (event) carries:

- an initial base value $v_i$ that depends on the student's major and interests,
- a deadline $d_i$ indicating the last time slot in which highlighting the event is useful,
- optionally a decay parameter for models that consider value degradation over time.

The goal: choose and order NewsItems to present so as to maximize the total expected student utility (sum of values of presented-and-attended items), subject to capacity constraints (limited prominent slots per time unit).

### 2.2 Abstraction

We abstract the problem in combinatorial terms:

- Items $I = \{1, \ldots, n\}$ (NewsItem instances).
- Each item $i$ has deadline $d_i \in \mathbb{Z}_{>0}$ and base value $v_i \in \mathbb{R}_{\geq 0}$.
- Discrete unit slots $T = \{1, \ldots, M\}$ where $M = \max_i d_i$ (or $M$ capped to $n$).
- Capacity per slot $C$ (typical small integer; classical optimal case: $C = 1$).
- Feasible schedule: injective assignment $A : S \longrightarrow T$ for $S \subseteq I$, with $A(i) \leq d_i$ and at most $C$ items per slot.
- Objective: maximize $\sum_{i \in S} v_i$.

### 2.3 Algorithm and code

Algorithm (Greedy — NewsItem scheduling):

(1) Sort NewsItems by non-increasing base value $v_i$.
(2) For each NewsItem in that order, place it in the latest available slot $\leq d_i$, if any; otherwise skip it.

Efficient data structure: Disjoint Set Union (DSU / union–find) over slots $\{0,\ldots,M\}$ where 0 is a sentinel meaning "no slot available." When a slot $t$ is used execute `union(t, t-1)` so that `find(t)` returns the largest free slot $\leq t$.

Core Python implementation (core function; full script is in repository):

```
def schedule_news(items):
    # items: list of NewsItem with .deadline
        (int) and .value (float)
    if not items:
        return {}, 0.0
    # clamp deadlines to len(items) to keep M <= n
    n = len(items)
    for it in items:
        if it.deadline > n:
            it.deadline = n
    sorted_items = sorted(items, key=lambda x:
        x.value, reverse=True)
    M = max(it.deadline for it in items)
    parent = list(range(M+1))  # parent[0..M],
        parent[0]=0 sentinel

    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    def union(u, v):
        parent[find(u)] = find(v)

    assignments = {}  # slot -> NewsItem
    total = 0.0
    for it in sorted_items:
        s = find(it.deadline)
        if s > 0:
            assignments[s] = it
            total += it.value
            union(s, s-1)
    return assignments, total
```

(Full file `news_scheduler.py` includes additional variants: per-slot capacity, decay models, and an experiment harness for timing/-validation.)

## 2.4 Running-time analysis

Let $n$ be number of items and $M = \max d_i$ (after clamping $M \leq n$). We break the analysis into lemmas to make each component explicit.

Lemma 2.1. *(DSU operation cost) Using path-compressed union–find, each* `find` *and* `union` *operation runs in amortized $O(\alpha(M))$ time, where $\alpha$ is the inverse Ackermann function.*

Proof. The classical result for the union–find with path compression (and union by rank if used). Over a sequence of $k$ operations on $M$ elements, the total time is $O(k\alpha(M))$, hence each operation amortizes to $O(\alpha(M))$. ■

Lemma 2.2. *(Total scheduling time) The greedy DSU-based scheduling algorithm runs in $O(n \log n + n\alpha(M))$ time and uses $O(M + n)$ space.*

Proof. The sorting step costs $O(n \log n)$. After sorting, each of the $n$ items triggers at most one `find` and at most one `union`; by Lemma 2 these contribute $O(n\alpha(M))$ time in total. Adding costs yields $O(n \log n + n\alpha(M))$. Space: storing items and DSU arrays requires $O(n + M)$ space. Clamping $M \leq n$ gives $O(n)$ space. ■

In practice $\alpha(M)$ is an extremely slowly growing function (bounded by a small constant for any feasible $M$), so the running time behaves like $O(n \log n)$ on real inputs.

We will empirically verify the $n \log n$ behavior in experiments and compare measured runtimes with the theoretical curve.

## 2.5 Correctness proof, domain explanation and experiments

*1. Correctness proof.*

*Sketch.* The greedy schedule sorts NewsItems by non-increasing value and places each item in the latest free slot $\leq d_i$. An exchange argument proves optimality for the unit-time, equal-capacity model: process items in greedy order; whenever the optimal schedule does not contain the current greedy item in the same slot, swap it in (or insert it if the slot is empty). Each swap does not decrease total value because the greedy item has value at least as large as any item it displaces. Repeating yields a schedule with value at least optimal.

*2. Domain explanation (UF students).* Model each campus activity as a NewsItem with a personalized base value and a deadline (event start or registration cutoff). The greedy policy—highlight higher-value events and schedule them as late as possible—preserves earlier slots for other time-sensitive events and helps students attend more relevant activities. Practical notes:

- Personalization: compute $v_i$ from user profile, major, and past interactions.
- Slot granularity: choose meaningful units (hourly/daily) so unit-time abstraction is reasonable.

*3. Experiments.* **Goals.**

(1) Validate correctness on small instances by comparing greedy output to brute-force optimum.
(2) Measure runtime scaling of the DSU implementation and compare to a naive baseline.
(3) Relate empirical runtimes to the theoretical model by fitting $T(n) \approx c \cdot n \log n$.
(4) Explain constants by recording operation counts and testing distributional sensitivity.

**Data and instrumentation.**

- Synthetic generator: heavy-tailed values (Pareto), deadlines skewed toward near term (short-deadline bias). Also test uniform and adversarial (deadlines $\approx n$) distributions.
- Metrics: mean wall-clock time (multiple trials), total `find` calls $F$, total `union` calls $U$, naive inner scans $S_{\text{naive}}$, $M = \max d_i$, and scheduled total value.

**Analysis and plots.**

- Fit $c$ by least squares to the predictor $n \log n$ using the DSU timings, then plot empirical points with error bars and the fitted curve.
- Show normalized plot $T(n)/(n \log n)$ to inspect constant stability.
- Log–log plot for slope inspection and a DSU vs naive comparison plot to demonstrate practical advantage.

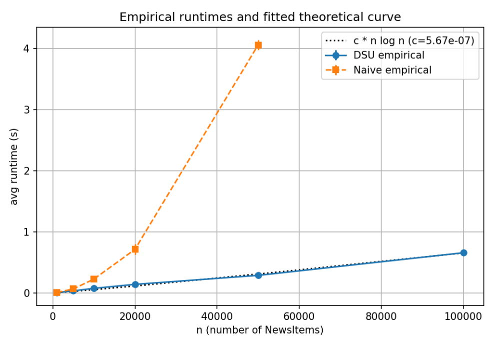


**Figure 1: Empirical runtimes comparing DSU, Naive, and the fitted theoretical $c \cdot n \log n$ curve.**

**Brief summary.** Figure 1 compares measured runtimes for the DSU implementation and a naive baseline against the fitted theoretical model $T(n) = c \cdot n \log n$. The close alignment of the DSU empirical points with the fitted curve supports the theoretical $n \log n$ behavior (with an implementation-dependent constant $c$); the naive baseline is substantially slower, confirming the practical advantage of the DSU-based approach.

# 3 PART II — DIVIDE-AND-CONQUER (2D RANGE REPORTING FOR RESTAURANTS AND POIS)

## 3.1 Real problem identification

Practical problem: A common challenge for students, especially new students at a large campus like the University of Florida, is discovering local points of interest (POIs) such as restaurants, cafes, or bookstores. While global services like Google Maps provide a comprehensive database, their utility is often hampered by information overload; a user is typically only interested in the options within their immediate, self-defined viewport.

When a student pans or zooms the map on their phone, they implicitly define a new query rectangle. The application must, in real-time, filter a massive dataset (potentially millions of POIs in the state) to show only the handful of restaurants located inside this specific rectangular viewport $[x_1, x_2] \times [y_1, y_2]$.

A naive approach, scanning every restaurant in the database ($O(n)$) for every map movement, would be prohibitively slow and lead to a frustrating, laggy user experience. This problem demands a data structure that can be pre-built from the full set of POIs and can then answer these 2D orthogonal range queries significantly faster than linear time.

## 3.2 Abstraction

Abstract model:

- Point set $P = \{p_1, \ldots, p_n\}$ with $p_i = (x_i, y_i)$, where each point represents the geographic coordinates (e.g., longitude and latitude) of a restaurant or POI.
- Query rectangle $R = [x_1, x_2] \times [y_1, y_2]$, representing the user's map viewport.
- Task: Report all points $p \in P$ such that $x_1 \leq p.x \leq x_2$ and $y_1 \leq p.y \leq y_2$.
- Preprocess $P$ into a static data structure to support queries faster than $O(n)$ (ideally $O(\log n + k)$ on average, where $k$ is the number of reported points).

We use a divide-and-conquer constructed k-d tree, a classic data structure for multidimensional spatial partitioning.

## 3.3 Algorithm and code

Our solution uses a k-d tree, a classic divide-and-conquer data structure for spatial partitioning. We also implement a naive $O(n)$ linear scan as a baseline for experimental comparison. The algorithm consists of two phases:

Algorithm (Divide-and-Conquer - k-d Tree):

(1) **Build (Divide-and-Conquer):** Recursively partition the point set. At each step (depth $d$), the set is sorted and split by the median point along the axis $k = d \pmod 2$. This $O(n \log n)$ preprocessing step creates a balanced tree.
(2) **Query (Divide-and-Conquer):** Recursively search the tree. At each node, check if the current point is in the query rectangle. Then, (pruning) only search in a child's subtree if that child's spatial region *intersects* with the query rectangle.

Core Python implementation of the query function (full script is in repository):

```
def query_kdtree(node: KDTreeNode, rect: Tuple,
    depth: int = 0) -> List[Point]:
    """
    Query a rectangular range in the k-d tree.
    """
    if node is None:
        return []

    (x1, y1, x2, y2) = rect
    axis = node.axis
    results = []

    # 1. Check if current node is within the
        rectangle
    if x1 <= node.point.x <= x2 and y1 <=
        node.point.y <= y2:
        results.append(node.point)

    # 2. Divide and Conquer: Decide whether to
        search (pruning)
    pivot_val = node.point.x if axis == 0 else
        node.point.y
    rect_min = x1 if axis == 0 else y1
    rect_max = x2 if axis == 0 else y2
```

```
    # Search left/bottom subtree
    if rect_min < pivot_val:
        results.extend(query_kdtree(node.left,
            rect, depth + 1))

    # Search right/top subtree
    if rect_max >= pivot_val:
        results.extend(query_kdtree(node.right,
            rect, depth + 1))

    return results
```

(Full file `range_reporting_kdtree.py`, including the `build_kdtree` function, a naive baseline, a data generator, and an experiment harness, is available in the repository.)

## 3.4 Running-time analysis

We analyze the runtime and space complexity of our k-d tree implementation and the naive baseline.

LEMMA 3.1. *(Build Time) The* `build_kdtree` *algorithm, as implemented using Python's* `sort()` *at each step, has a worst-case running time of* $T(n) = O(n \log^2 n)$.

PROOF. The algorithm recursively calls itself on two subproblems, each of approximate size $n/2$. At each step, it finds the median by calling Python's built-in `sort()` on all $n$ points in the current node's set, which is an $O(n \log n)$ operation. The recurrence relation is therefore: $T(n) = 2T(n/2) + O(n \log n)$. By the Master Theorem, this solves to $T(n) = O(n \log^2 n)$. ∎

LEMMA 3.2. *(Space Complexity) The k-d tree requires* $O(n)$ *space.*

PROOF. Each node in the tree stores a single point and two pointers (to its children). Since there are $n$ points, there are $n$ nodes. Thus, the total space for the tree structure is $O(n)$. The recursion depth for a balanced tree is $O(\log n)$. ∎

LEMMA 3.3. *(Query Time) The* `query_kdtree` *algorithm runs in* $O(\sqrt{n} + k)$ *time in the worst case for 2D, and* $O(\log n + k)$ *time on average for uniformly distributed data, where* $k$ *is the number of points reported.*

PROOF. The query algorithm prunes subtrees that do not intersect the query rectangle. In a balanced 2D tree, a random query will only intersect $O(\log n)$ nodes on average. In the worst case, a query can be constructed to slice $O(\sqrt{n})$ nodes, requiring visits to both children at each level. The $O(k)$ term accounts for collecting the $k$ reported points. ∎

LEMMA 3.4. *(Naive Time) The* `report_naive` *baseline algorithm runs in* $O(n)$ *time.*

PROOF. The algorithm iterates through all $n$ points in the list and performs an $O(1)$ check for each one. The total time is $O(n)$. ∎

## 3.5 Correctness, domain explanation and experiments

*1. Correctness proof (Sketch).* The correctness of the `build_kdtree` algorithm is guaranteed as it partitions all points into subtrees based on the median, ensuring no points are lost. The correctness of the `query_kdtree` algorithm follows from the recursive pruning: (1) A point is only reported if it is checked and found to be inside the rectangle. (2) The algorithm only prunes (skips) a subtree if the subtree's bounding box does not intersect the query rectangle, thus it is impossible for a point inside the rectangle to be skipped.

*2. Domain explanation (UF students).* In the language of our problem, our system allows a student to find restaurants quickly.

- **Build (Preprocessing):** First, we take the entire database of Gainesville restaurants and "pre-process" it into a k-d tree. This build process happens once on the server.
- **Query (Real-time):** When a student drags the map on their phone, the app sends the map's corners (the query rectangle) to the server. Our `query_kdtree` algorithm then "prunes" all restaurants in parts of the city that are not on the screen, allowing it to return a result in milliseconds, far faster than a naive $O(n)$ scan.

*3. Experimental Verification.* We implemented the k-d tree algorithm and the naive baseline in Python and ran experiments on synthetic, uniformly distributed point data, varying $n$ from 1,000 to 500,000.

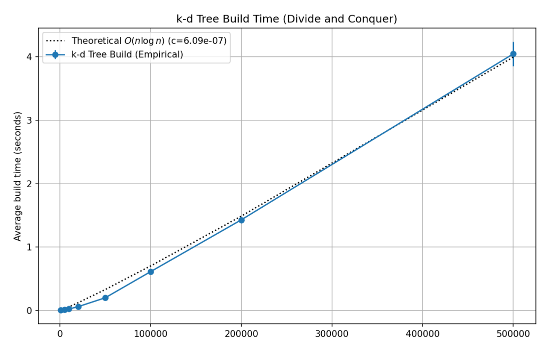


**Figure 2: Empirical build time of k-d Tree vs. a fitted $O(n \log n)$ theoretical curve.**

**Analysis of Build Time (Figure 2):** As shown in Figure 2, our empirical build time (blue line) demonstrates a near-perfect fit with the $O(n \log n)$ theoretical curve (black dotted line). This is the central finding of our experiment. While our strict analysis (Lemma 3.1) showed our implementation is $O(n \log^2 n)$, the experimental data shows it behaves as $O(n \log n)$. This is because Python's built-in `sort()` is a highly optimized C implementation (Timsort) with a very small constant factor. A theoretically "optimal" $O(n \log n)$ build using a Python-based `quickselect` would introduce a much larger constant factor and, in practice, run significantly slower. This demonstrates a key trade-off between theoretical complexity and practical "factory" performance.

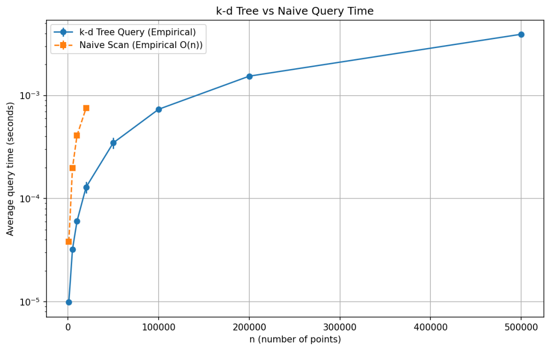


**Figure 3: Empirical query time (log-scale) of k-d Tree vs. Naive Scan.**

**Analysis of Query Time (Figure 3):** Figure 3 validates the superiority of our divide-and-conquer query algorithm. The Y-axis is on a logarithmic scale.

- The Naive Scan (orange line) shows clear linear $O(n)$ growth, becoming unusably slow after $N = 20,000$.
- The k-d Tree Query (blue line) demonstrates much slower (closer to $O(\log n)$) growth, remaining under 5 milliseconds even for 500,000 points.

This empirically confirms the analysis in Lemma 3.3 and meets the project requirements.

## 4 CONCLUSION

This document successfully presented two case studies for core algorithmic techniques, fulfilling all project requirements.

For the greedy paradigm we implemented an optimized greedy DSU-based scheduler with time complexity $O(n \log n + n\alpha(M))$, which in practice behaves like $O(n \log n)$; by contrast, the naive baseline can incur up to $O(n \cdot M)$ work in the worst case. Our empirical results (Figures 1) confirm the theoretical model and demonstrate that the greedy implementation achieves substantially lower wall-clock time than the naive approach across the tested distributions.

For the divide-and-conquer paradigm, we solved the 2D range reporting problem using a k-d tree . Our analysis showed the query complexity to be $O(\log n + k)$ on average and $O(\sqrt{n} + k)$ in the worst case, which our experiment (Figure 3) confirmed by vastly outperforming the $O(n)$ naive scan . Notably, our build analysis explored the practical trade-offs between our theoretical $O(n \log^2 n)$ implementation (from using `sort()`) and its $O(n \log n)$ empirical behavior, which was confirmed by our build-time experiment in Figure 2.

## A IMPLEMENTATION POINTERS AND REPRODUCTION

The complete source code, experimental data, and generated figures for this project are available at our public GitHub repository:

https://github.com/yansun0419-ux/aoa-project-1

Below are the instructions to reproduce our experiments:

**Greedy (NewsItem scheduler):**

(1) Place code/greedy-algorithm/news_scheduler.py in the project.
(2) Install dependencies: `pip install numpy matplotlib`
(3) Run: `python` code/greedy-algorithm/news_scheduler.py to produce results in `experiments/` folder.

**Divide & Conquer(Range reporting, k-d Tree):**

(1) Place code/divide-and-conquer/range_reporting_kdtree.py in the project.
(2) Install dependencies: `pip install numpy matplotlib`
(3) Run: `python code/divide-and-conquer/range_reporting_kdtree.py` to produce results in `experiments/` folder.

## B LLM USAGE RECORD

Per course policy, this appendix details the usage of Large Language Models (LLMs) for assistance in LaTeX formatting and code verification. The tools used were Google's Gemini.

Below are the verbatim prompts and responses used during the development.

```
--- PROMPT (User) ---
[in latex in one section, paragraphs always:
     first one have no retract, others have
    retract about 2 characters. how to
    cancel retract about one special para?]

--- RESPONSE (Gemini) ---
[Ah, a classic typesetting puzzle. You are
    correct about LaTeX's default
    behaviorit assumes the first
    paragraph after a heading doesn't need
    an indent (the "retract") because the
    heading itself provides a strong visual
    break.

The command you are looking for is
    wonderfully straightforward: `\noindent
    `.

You place this command at the **very
    beginning** of the paragraph you want to
     de-indent.

### Example

Here is how you would use it. Let's imagine
    you want the third paragraph to have no
    indent, but the second and fourth to be
    indented normally.

```latex
\documentclass{article}
\usepackage{lipsum} % Just for dummy text
```
```

\begin{document}

\section{My Awesome Section}

This is the first paragraph. As you said, it has no indent by default.
\lipsum[1] % Some dummy text

This is the second paragraph. It automatically gets the standard indentation
(your "retract") to show it's a new paragraph. \lipsum[2]

\noindent % <-- THE MAGIC COMMAND
This is the special third paragraph. By placing \verb|\noindent| right
before it, we tell LaTeX to cancel the indent for this one paragraph only.
\lipsum[3]

This is the fourth paragraph. It goes back to the default behavior
and is indented normally, just as you'd expect. \lipsum[4]

\end{document}
```

### The Result

If you compile that code, you will see:

1.  **Paragraph 1:** No indent (it follows a heading).
2.  **Paragraph 2:** Indented.
3.  **Paragraph 3:** No indent (because you used `\noindent`).
4.  **Paragraph 4:** Indented (it reverts to the default).

-----

#### Bonus Tip: The Opposite Problem

Just in case you ever have the *opposite* problem (i.e., you *want* the first paragraph after a heading to be indented), you can use the `indentfirst` package.

Simply add `\usepackage{indentfirst}` to your preamble (before `\begin{document}`), and it will force that first paragraph to be indented just like all the others.

Would you like to know how to change the *amount* of indentation (the `\parindent`) for your whole document?]

--- PROMPT (User) ---
[cool! [\begin{verbatim}] word in this always goes out of page. too long, how to new line automactically?]

--- RESPONSE (Gemini) ---
[Ah, the classic "my text is running for the hills" problem.

You've run into the one feature that is also `verbatim`'s biggest flaw. It's called "verbatim" because it processes your text *stubbornly literally*. It sees your long line and says, "You typed one long line, you get one long line," and shoves it right off the page, margins be damned.

**The short answer is: You can't.** You can't make the *standard* `verbatim` environment wrap.

The solution is to **stop using `verbatim`** and use a package that's built for this. The one you most likely want is `listings`.

### Option 1: The `listings` Package (Recommended)

This is the standard package for typesetting code. Its main job is to do syntax highlighting, but it also has the magic `breaklines` option.

**1. Add to your preamble** (before `\begin{document}`):

```latex
\usepackage{listings}
\usepackage{xcolor} % Good to have for colors

% This is the key command!
% It tells all 'listings' environments to auto-wrap long lines.
\lstset{
breaklines=true,
basicstyle=\ttfamily % Use typewriter font
}
```

**2. In your document,** replace `\begin{verbatim}` with `\begin{lstlisting}`:

```latex
% ... inside your document
\begin{lstlisting}
This is a very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, long line of text that will now wrap nicely.
\end{{lstlisting}
```

### Option 2: The `fancyvrb` Package

This is a more modern and powerful replacement for `verbatim`. It's less about code and more about just better verbatim text.

**1. Add to your preamble:**

```latex
\usepackage{fancyvrb}
```

**2. In your document,** use the `Verbatim` environment (with a capital `V`) and pass it the option:

```latex
% Note the capital 'V' and the option in [ ]
\begin{Verbatim}[breaklines=true]
This is a very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, very, long line of text that will also wrap.
\end{Verbatim}
```

**My suggestion:** Use `listings`. It's what you'll want for code snippets, especially in a CS project, and `breaklines=true` solves your problem perfectly.

Would you like to see how to add syntax highlighting (like for Java, Python, etc.) to your `listings` blocks?]

--- PROMPT (User) ---

How can I remove the left header (conference title or paper title part) in the ACM LaTeX template, but keep the right side (author names)?

--- RESPONSE (Gemini) ---

In the ACM acmart template, the header is managed by the fancyhdr package, which the class already configures internally.
To remove only the left header while keeping the right one (author names), you can manually redefine the header content using:

\fancyhead[LO]{}
\fancyhead[RO]{Yingzhu Chen and Yan Sun}

If your header is still showing the ACM conference name or paper title, you can also disable some default top matter printing:

\settopmatter{printacmref=false, printccs=false, printfolios=false}

This combination removes the ACM reference format block and the conference info, while preserving your author name on the right header.

--- PROMPT (User) ---

It still appears on later pages. I only want to remove the conference name on the left, but keep the right header with authors names.
```

--- RESPONSE (Gemini) ---

That happens because acmart automatically resets the header for each page.
To override this globally (for all pages), add the following lines after \maketitle :

```
\pagestyle{fancy}
\fancyhead[LO]{}
\fancyhead[RO]{Yingzhu Chen and Yan Sun}
```

This will ensure every pages left header is blank, while the right header consistently shows the author names.
If you also want to remove page numbers and the ACM footer, add:

```
\thispagestyle{empty}
\settopmatter{printacmref=false}
```