1. Background Introduction

1.1 Texture Packing Problem

- Texture Packing is a problem where rectangles of various sizes need to be packed into a container with certain width.
- The goal is to minimize the total height of the container while fitting all rectangles without overlap at the same time.

1.2 Existing Algorithms

- We implemented two heuristic algorithms: NFDH (Next Fit Decreasing Height) and FFDH (First Fit Decreasing Height).
- Both algorithms sort rectangles by height in decreasing order before packing.
- The difference lies in how they place rectangles: NFDH always places the next rectangle in the current level, while FFDH searches for the first level where the rectangle fits.

1.3 Performance Considerations

- The efficiency of these algorithms depends on the number of rectangles and the container width (binWidth).
- We aim to compare both the packing efficiency (height) and computational performance (time) of these algorithms.

2. Experiments and Performance Evaluation

2.1 Experiments Procedure

- We conducted experiments with three different binWidth values: 1000, 2000, and 5000.
- For each binWidth, we tested with varying numbers of rectangles: 1000, 3000, 5000, 8000, 10000, 30000, and 50000.
- We measured both the total height achieved by each algorithm and the execution time.
- All experiments were run on the same hardware configuration to ensure fair comparison.

2.2 Tables and Graphs of Results

Height Performance Comparison

binWidth	Number of Rectangles (n)	NFDH Height	FFDH Height	Height Improvement (%)
1000	1000	95888	77855	18.81
1000	3000	299953	239352	20.20
1000	5000	492235	389205	20.92
1000	8000	806917	625385	22.50
1000	10000	1016408	784821	22.78
1000	30000	3109031	2330112	25.05
1000	50000	5198569	3878967	25.38

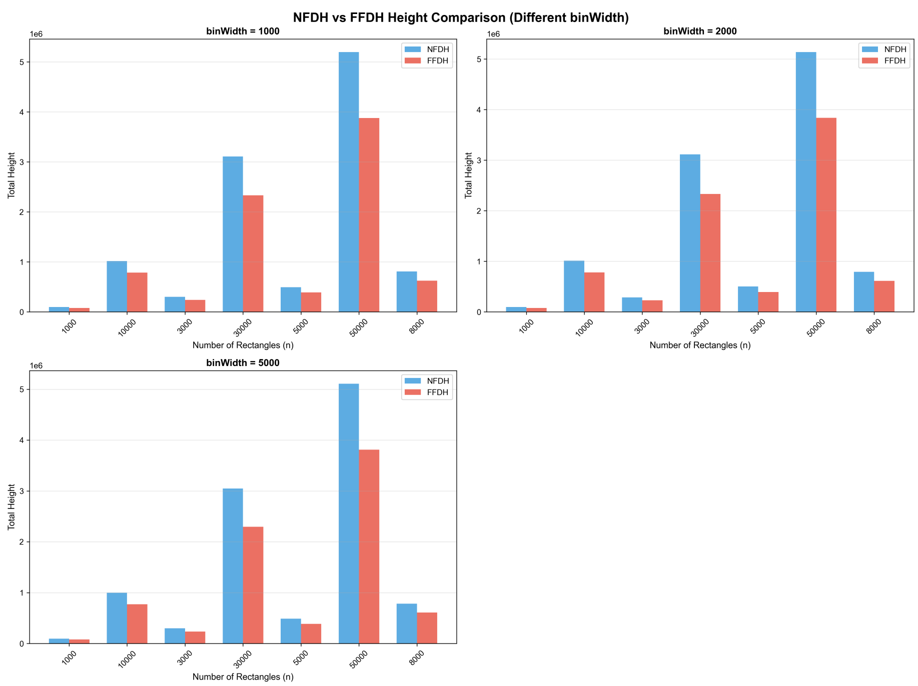

Figure 1: Height Comparison

Time Performance Comparison

binWidth	Number of Rectangles (n)	NFDH Time (ms)	FFDH Time (ms)
1000	1000	0.000	0.000
1000	3000	0.000	1.000
1000	5000	0.000	2.000

binWidth	Number of Rectangles (n)	NFDH Time (ms)	FFDH Time (ms)
1000	8000	1.000	2.000
1000	10000	1.000	3.000
1000	30000	1.000	32.000
1000	50000	3.000	95.000

FFDH Improvement Over NFDH

2.3 Performance Evaluation & Analysis

Height Performance

- FFDH consistently outperforms NFDH in terms of packing efficiency across all binWidth values and rectangle counts.
- The height improvement percentage ranges from 18.81% to 25.38%, showing that FFDH achieves better space utilization.
- The improvement percentage generally increases with the number of rectangles, suggesting that FFDH scales better with larger problem instances.

Time Performance

- NFDH is generally faster than FFDH, especially for larger numbers of rectangles.
- The time difference becomes more significant as the number of rectangles increases.
- For small instances ($n \leq 1000$), both algorithms have negligible execution time.
- For large instances ($n = 50000$), FFDH takes significantly longer (95ms vs 3ms for NFDH with binWidth=1000).

Trade-off Analysis

- There is a clear trade-off between packing efficiency and computational speed.
- FFDH provides better packing efficiency (lower height) at the cost of increased computational time.
- NFDH is faster but produces less optimal packings (higher height).
- The choice between algorithms depends on the specific requirements: if space is critical, FFDH is preferable; if speed is prioritized, NFDH is better.

Effect of binWidth

- The binWidth parameter affects both algorithms' performance.
- Larger binWidth values generally result in lower total heights for both algorithms.

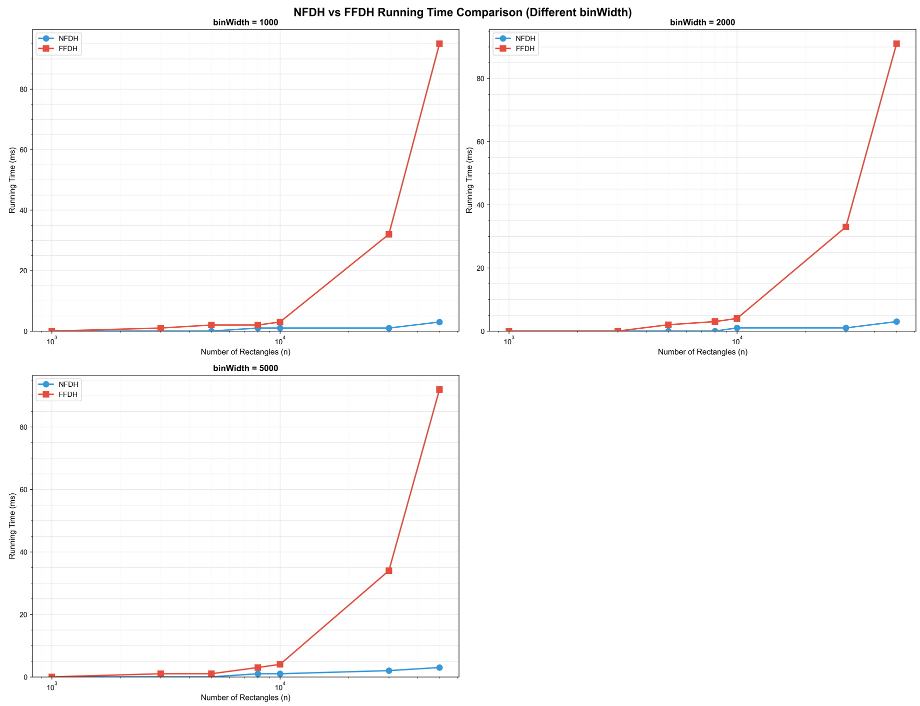

Figure 2: Time Comparison

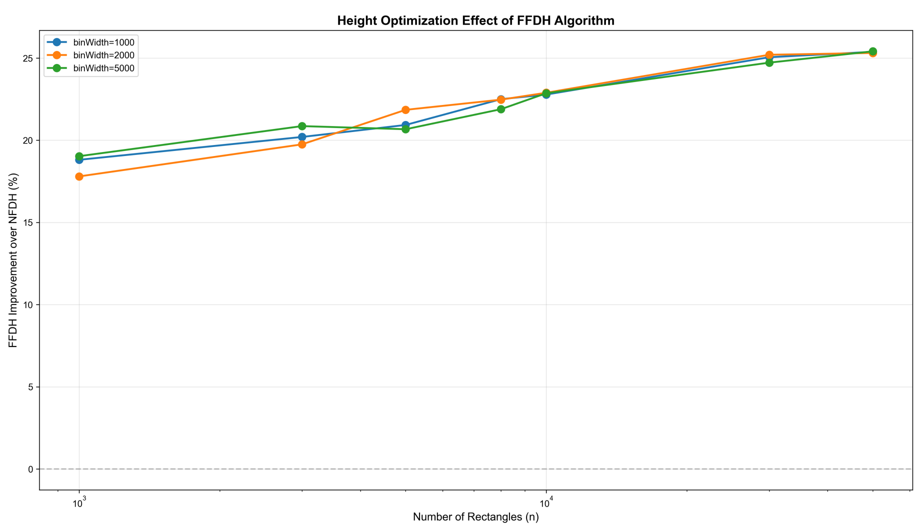

Figure 3: Improvement Percentage

- The relative performance difference between NFDH and FFDH remains consistent across different binWidth values.

Theoretical vs. Observed Complexity:

Algorithm	Time Complexity	Height Efficiency	Observed Behavior
NFDH	O(n log n)	Moderate	Fast execution, higher height
FFDH	O(n^2)	High	Slower execution, lower height

3. Conclusions

- FFDH consistently achieves better packing efficiency than NFDH, with height improvements ranging from 18.81% to 25.38%.
- The computational cost of FFDH is significantly higher than NFDH, especially for large problem instances.
- The choice between NFDH and FFDH depends on the specific application requirements: space efficiency vs. computational speed.
- For applications where packing efficiency is critical (e.g., minimizing material usage), FFDH is the preferred choice despite its higher computational cost.
- For applications requiring fast packing with moderate space efficiency, NFDH is more suitable.
- The binWidth parameter affects both algorithms' performance, with larger widths generally resulting in better space utilization.

Appendix: Source Code in CPP

NFDH Algorithm

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <iomanip>
using namespace std;

struct Rectangle {
    int id;
    int width;
    int height;
};
// the struct to store the features of rectangle
```

```
bool cmpHeight(const Rectangle &a, const Rectangle &b) {
    if (a.height != b.height) return a.height > b.height;
    return a.width > b.width;
    // Helper function to compare the heights of rectangles
}

// NFDH Algorithm: Next Fit Decreasing Height
int nfdh(vector<Rectangle> &rects, int binWidth) {
    sort(rects.begin(), rects.end(), cmpHeight);

    int currentShelfUsedWidth = 0;
    int currentShelfHeight = 0;
    int totalHeight = 0;

    for (size_t i = 0; i < rects.size(); ++i) {
        Rectangle &r = rects[i];

        if (currentShelfUsedWidth + r.width <= binWidth) {
            currentShelfUsedWidth += r.width;
            currentShelfHeight = max(currentShelfHeight, r.height);
            // if the current rectangle can fit the width, then fit it and update our heigh
        } else {
            if (currentShelfHeight > 0) {
                totalHeight += currentShelfHeight;
                // we need to add another shelve
            }
            currentShelfUsedWidth = r.width;
            currentShelfHeight = r.height;
        }
    }

    if (currentShelfHeight > 0) {
        totalHeight += currentShelfHeight;
    }

    return totalHeight;
}

// FFDH: First Fit Decreasing Height
int ffdh(vector<Rectangle> &rects, int binWidth) {
    sort(rects.begin(), rects.end(), cmpHeight);

    // the array to store shelves
    vector<pair<int, int>> shelves; // (height, usedWidth)
```

```
    for (size_t i = 0; i < rects.size(); ++i) {
        Rectangle &r = rects[i];

        bool placed = false;

        // First Fit: we start from the first shelve to find the first fitting position
        for (size_t j = 0; j < shelves.size(); ++j) {
            if (shelves[j].second + r.width <= binWidth) {
                // this means that we can fit in the current shelve
                shelves[j].second += r.width;
                shelves[j].first = max(shelves[j].first, r.height);
                //update the
                placed = true;
                break;
            }
        }

        // if it cant be placed now, then we need to create another shelve
        if (!placed) {
            shelves.push_back({r.height, r.width});
        }
    }

    // calculate the total height
    int totalHeight = 0;
    for (size_t i = 0; i < shelves.size(); ++i) {
        totalHeight += shelves[i].first;
    }

    return totalHeight;
}

int main()
{
    int binWidth, n;
    cin >> binWidth >> n;

    vector<Rectangle> rects;
    for (int i = 0; i < n; ++i) {
        int w, h;
        cin >> w >> h;
        rects.push_back({i + 1, w, h});
    }

    clock_t start, end;
```

```
    // execute the NFDH Algorithm
    vector<Rectangle> rectsNFDH = rects;
    start = clock();
    int heightNFDH = nfdh(rectsNFDH, binWidth);
    end = clock();
    double timeNFDH = double(end - start) / CLOCKS_PER_SEC * 1000.0;

    // execute the FFDH Algorithm
    vector<Rectangle> rectsFFDH = rects;
    start = clock();
    int heightFFDH = ffdh(rectsFFDH, binWidth);
    end = clock();
    double timeFFDH = double(end - start) / CLOCKS_PER_SEC * 1000.0;

    // Output the results
    cout << "NFDH: " << heightNFDH << " (Time: " << fixed << setprecision(6) << timeNFDH <<
    cout << "FFDH: " << heightFFDH << " (Time: " << fixed << setprecision(6) << timeFFDH <<

    return 0;
}
```