浙江大学

实验报告

Project 6 Texture Packing

2024 年 11 月 27 日

Chapter 1

In this chapter, we will provide a description of the problem and an initial approach to solving it.

Problem Description

The **Texture Packing problem** can be described as follows:

Given a set of N rectangles, each with a width and height of (w_i, h_i), we have a rectangular stack box with a fixed width W. The goal is to pack these rectangles into the box in a way that minimizes the total height. All dimensions w_i, h_i, and W are integers.

Formally, the constraint requires that at any given height, the total width of rectangles at that level must not exceed W.

Ideas

This problem is known as the **2D Strip Packing** problem, which is an **NP-hard** problem. This means that to solve it within an acceptable amount of time, we must rely on approximation algorithms. After researching related materials online, we decided to use the **FFDH** (*First-Fit Decreasing Height*) algorithm to tackle this problem and compare its performance with the **NFDH** (*Next-Fit Decreasing Height*) and **BFDH** (*Best-Fit Decreasing Height*) algorithms.

Chapter 2

In this chapter, we will discuss how to analyze and solve this problem using dynamic programming in three steps.

First, let us review the one-dimensional version of this problem: we are given many bins, each with a capacity of 1, and the goal is to use the minimum number of bins to pack all items. However, in the current **2D Strip Packing** problem, we do not have such predefined "bins" to divide each stage, and we cannot simply assign items to individual bins. This makes the problem more challenging than its one-dimensional counterpart.

To address this, approximation algorithms introduce an important structure: **layers**. We assume that all rectangles in a layer have their bottom edges aligned on the same horizontal line, effectively dividing the packing rectangle into several layers. Each layer has two key properties:

1. The total width of all rectangles in the layer is $\leq W$.
2. The height of the layer equals the height of the tallest rectangle in that layer.

Formally, for the set of rectangles in the i-th layer, denoted as S_i, and for each rectangle j within S_i, the following conditions hold:

1. $\sum_{j \in S_i} w_j \leq W$, where w_j is the width of rectangle j.
2. $H_i = \max_{j \in S_i} h_j$, where H_i represents the height of the i-th layer, and h_j is the height of rectangle j.

The total height of the packing is then given by:
$H = \sum_i H_i$

With this structure, we can proceed to implement the **First-Fit** algorithm. The algorithm is described as follows:

For the current rectangle being considered:

1. Search through the existing layers to find the first layer where the remaining width is greater than or equal to the rectangle's width.
2. Place the rectangle in that layer.

If no such layer is found, create a new layer and place the rectangle there.

The pseudocode can be described as follows:

```
FOR i = 1 to n do
    found = false
    FOR j = 1 to cur do
        IF (width_left[j] >= width[i]) then
            H[j] = max(H[j], height[i])
            width_left[j] -= width[i]
            found = true
            break
    end FOR

    IF (not found) then
        cur = cur + 1
        width_left[cur] = W - width[i]
        H[cur] = height[i]
    end IF
end FOR
```

Here:

- `width_left[j]` tracks the remaining width in layer j.
- `H[j]` stores the height of layer j.
- `cur` represents the current number of layers.
- `W` is the total width of the container.

Here, we also need to use the offline version of the **First-Fit** algorithm, which involves sorting all rectangles in **descending** order of **height** before processing them sequentially.

In practice, if we draw an analogy to the one-dimensional version, we would sort by **width**. However, since our goal is to minimize the total height, it is more appropriate to sort by height in descending order.

The main reason for this is that we want each layer to consist of rectangles with similar heights, which reduces wasted space within each layer. The worst-case scenario occurs when each layer contains one very tall rectangle along with several very short ones. In this case, the total height becomes the sum of the heights of the tallest rectangles in each layer, leading to inefficient packing.

Data Structure:

Here, we implemented a **max-heap** to extract the rectangle with the maximum height each time. The max-heap supports two operations: **insertion** and **extracting the maximum**.

- During insertion, the heap structure is maintained by performing an **up-heap operation** (sifting up).

- When extracting the maximum, the root node's information is returned. The root is then swapped with the last element, the last element is removed, and the heap structure is restored by performing a **down-heap operation** (sifting down) starting from the root.

Therefore, instead of sorting the entire list, we insert all elements into the heap and extract the maximum one at a time.

Chapter 3

In this chapter, we will test the correctness of our solution using small-scale data and evaluate its performance on large-scale data.

Testing result(for correctness)

Proving the correctness of these programs is not straightforward, as they are **approximation algorithms** and there is no standard answer to verify whether the resulting height is correct. However, we can use small-scale data and **manually analyze** the results to ensure that the algorithm is being executed correctly.

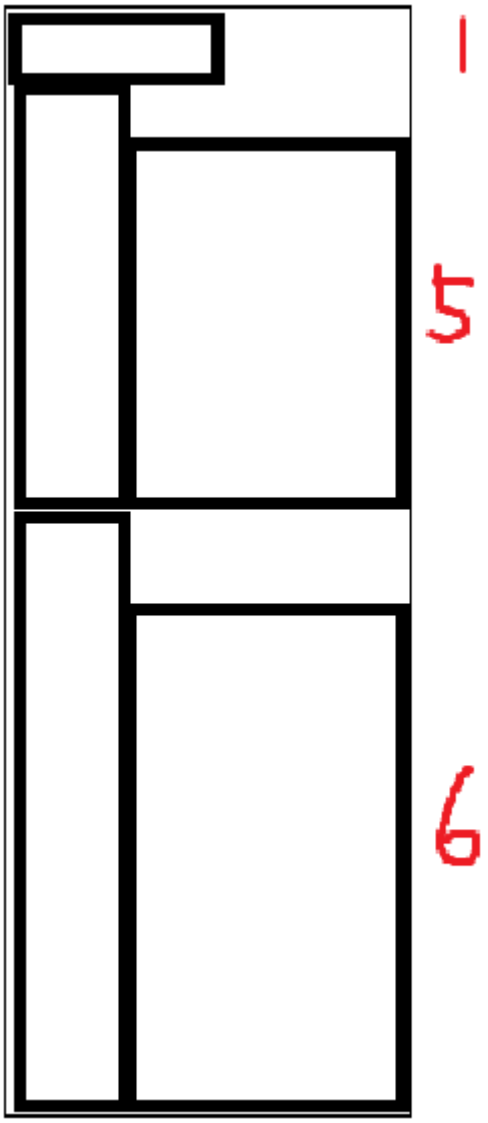
Input:

```
5 4
3 5
3 4
2 1
1 5
1 6
```

Output:

```
12
```

The result is as shown in the figure below, is the same as the algorithm should produce.



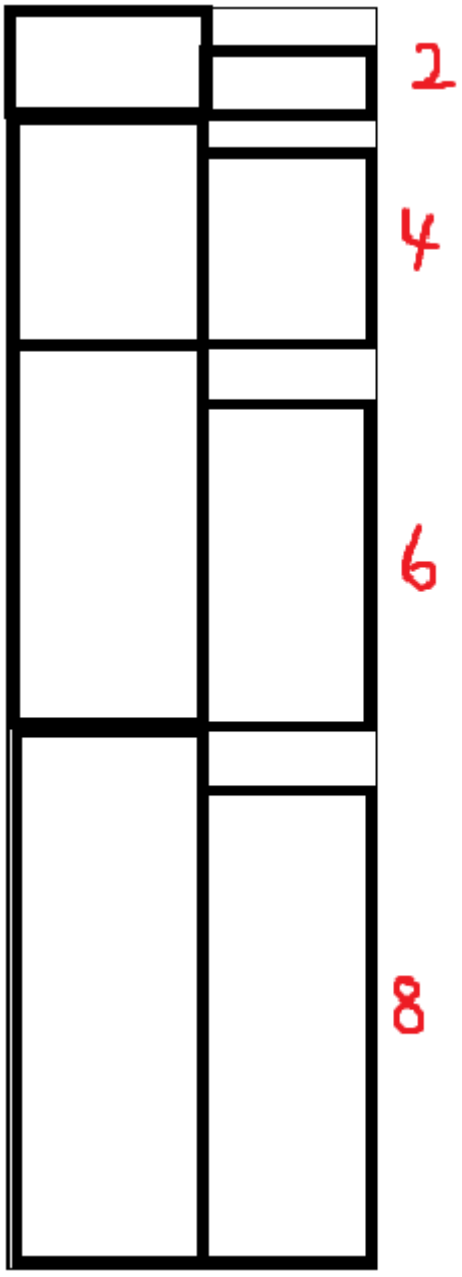
Input:

```
8 4
2 1
2 8
2 2
2 7
2 3
2 6
2 4
2 5
```

Output:

```
20
```

The result is as shown in the figure above, is the same as the algorithm should produce.



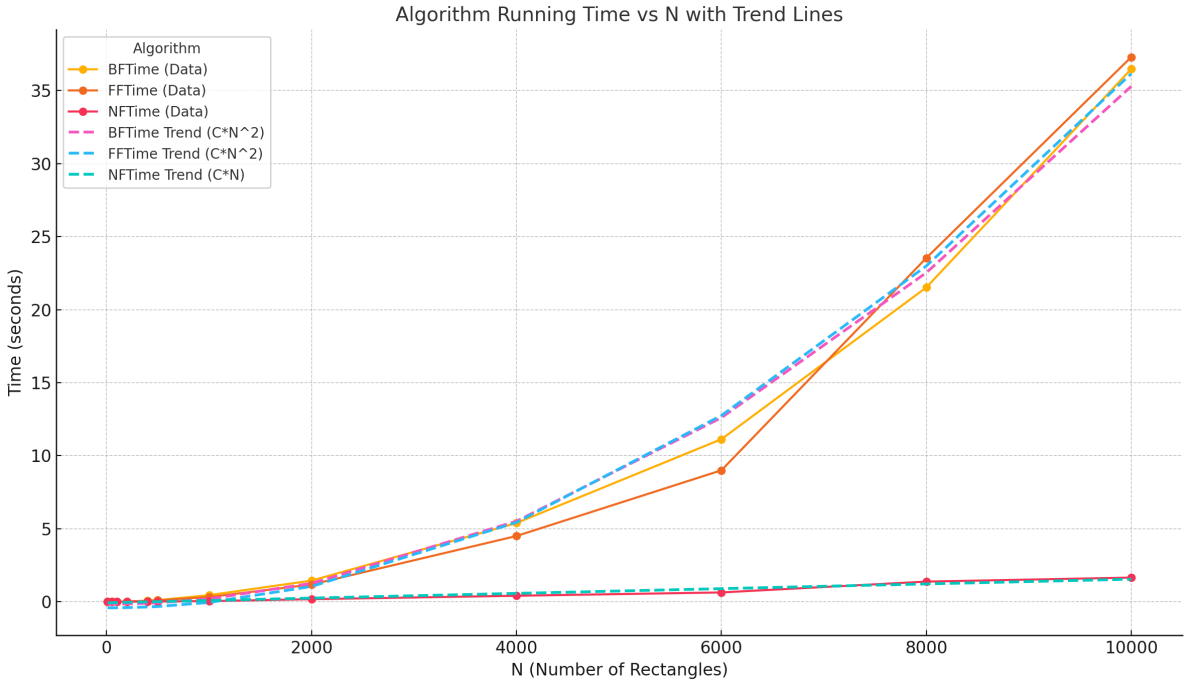
This dataset represents an **extreme case** where rectangles can only be stacked in pairs. Additionally, this dataset can test whether the program correctly sorts all rectangles in **descending order** of height before processing them offline. If rectangles are processed in the order of input instead, it will result in a very poor outcome.

Testing table(for efficiency and approximation ratio)

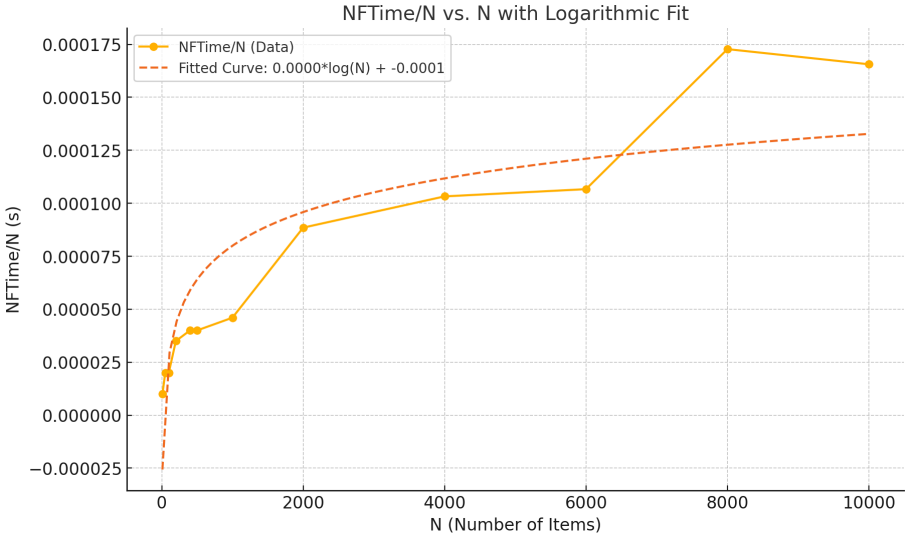
In this section, we begin to include the other two algorithms, **Next-Fit** and **Best-Fit**, in the comparison. We will compare their running times as well as their solutions' approximation ratio relative to a theoretical optimal baseline.

Time Table

N/W	10/10	50/30	100/100	200/120	400/200	500/300	1000/600	2000/1000	4000/3000	6000/4000	8000/7500	10000/8000
BFTime/s	0.0002	0.002	0.006	0.021	0.075	0.111	0.453	1.450	5.393	11.132	21.524	36.482
FFTime/s	0.0002	0.002	0.006	0.021	0.062	0.101	0.326	1.181	4.504	9.002	23.543	37.300
NFTime/s	0.0001	0.001	0.002	0.007	0.016	0.020	0.046	0.177	0.413	0.640	1.382	1.656



Based on the table and the graph, it can be preliminarily confirmed that the time complexity of **BF** and **FF** is approximately $O(N^2)$, while **NF** has a linear complexity of $O(N)$. We used the corresponding trend lines for fitting and ultimately validated this observation.



Based on this chart, we analyze the trend of **NF**'s runtime divided by N, and observe that it is not constant but increases as N grows, approximately following $O(\log N)$. Therefore, we can further confirm that the runtime of the **NF** algorithm is indeed $O(N \log N)$.

Performance Table

In this section, we will discuss how the solution quality of the **BF**, **FF**, and **NF** algorithms changes under input data with different distributions. We will further analyze their approximation ratios and attempt to explain these phenomena.

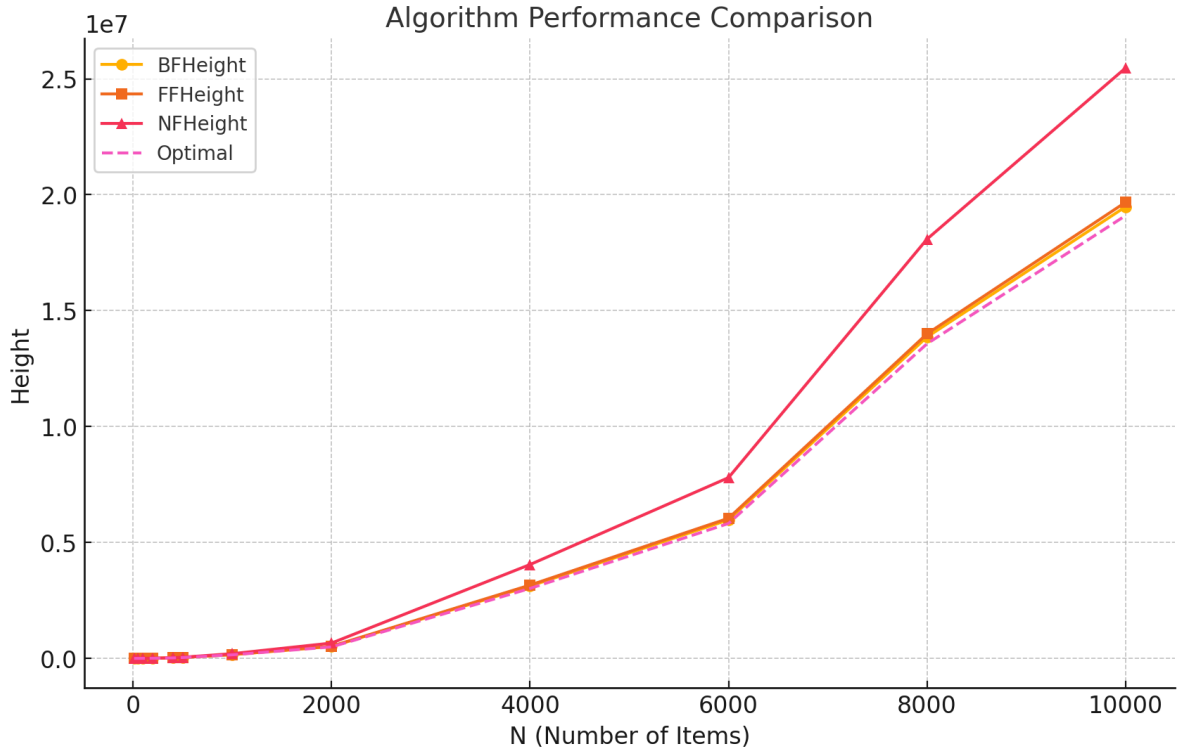
First, to determine the quality of the solutions provided by the three algorithms, we need to compare them to the optimal solution. However, the actual optimal solution cannot be obtained in polynomial time. Therefore, we use an approximation for the optimal solution:

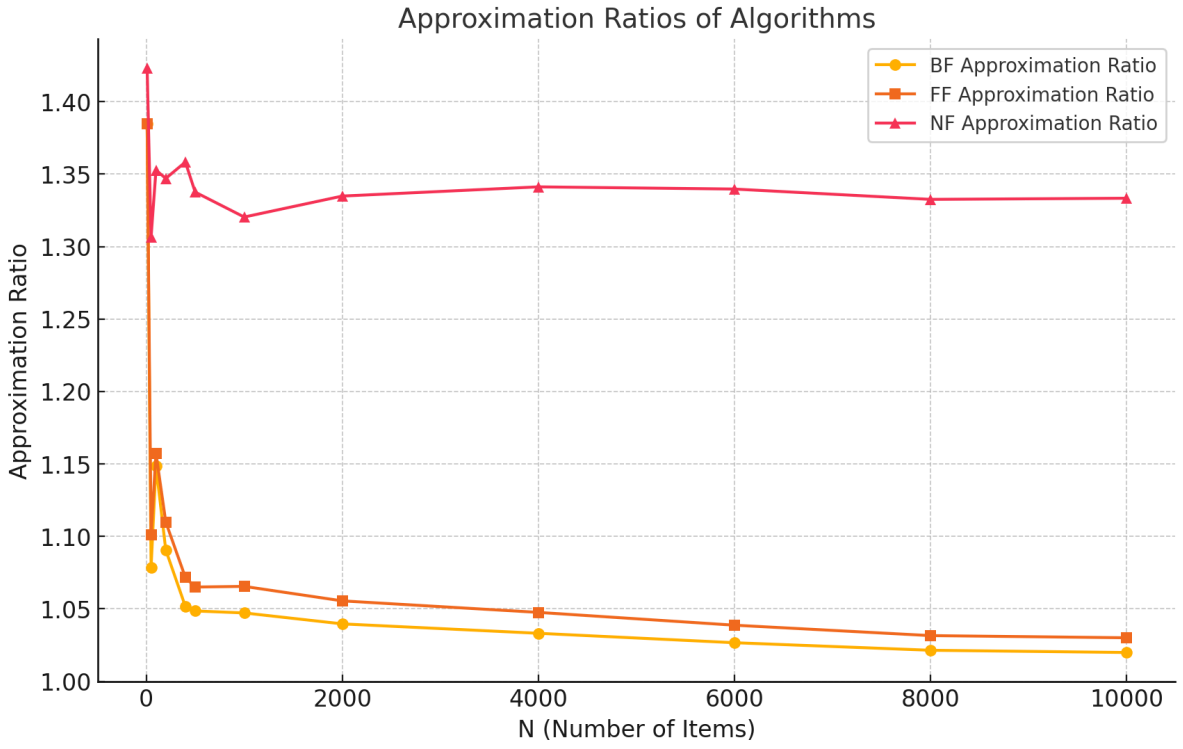
$$\mathrm{OPT} = \frac{1}{W} \sum (\mathrm{height}_i \times \mathrm{width}_i)$$

This represents the total area of all rectangles divided by the width constraint, serving as the ideal optimal solution.

Case 1: Fully Random

Here, we generate the width and height of the rectangles using completely random numbers, resulting in a fairly uniform upward trend in the outcomes.





Observing the approximation ratios, we can make the first observation:

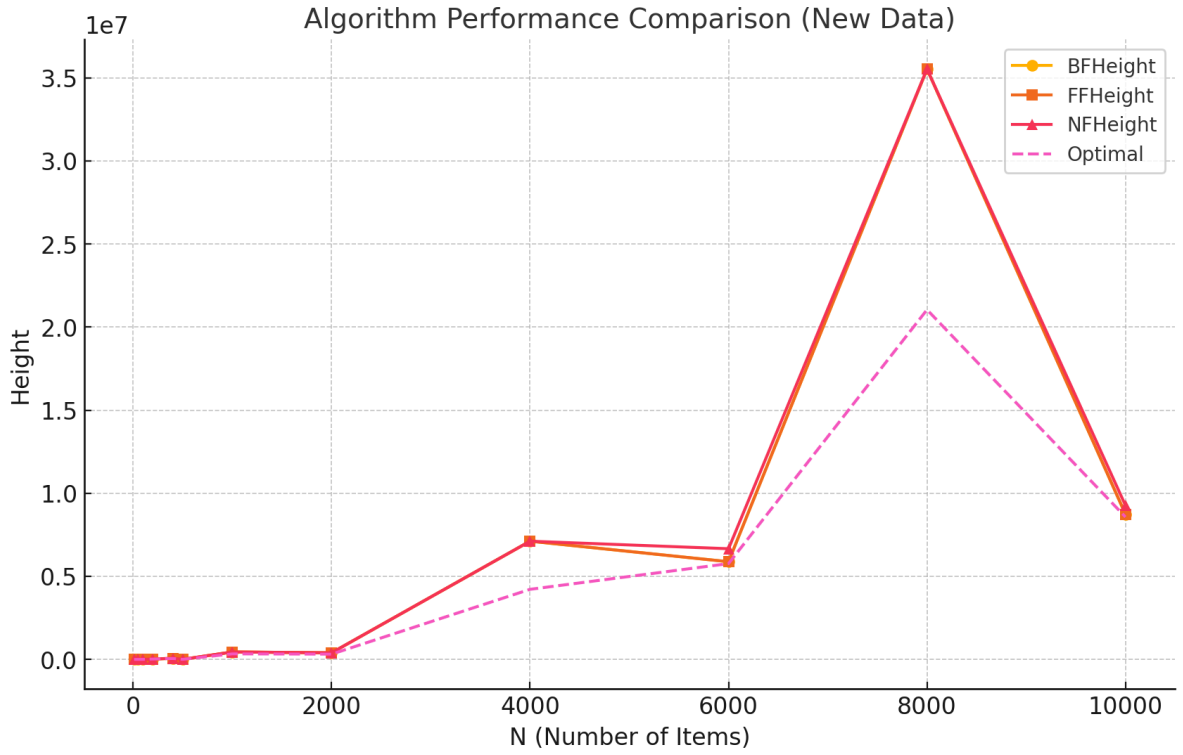
When N is relatively small, the combination of different rectangles can cause significant fluctuations in the approximation ratio. However, as N becomes sufficiently large, the approximation ratios stabilize. Specifically:

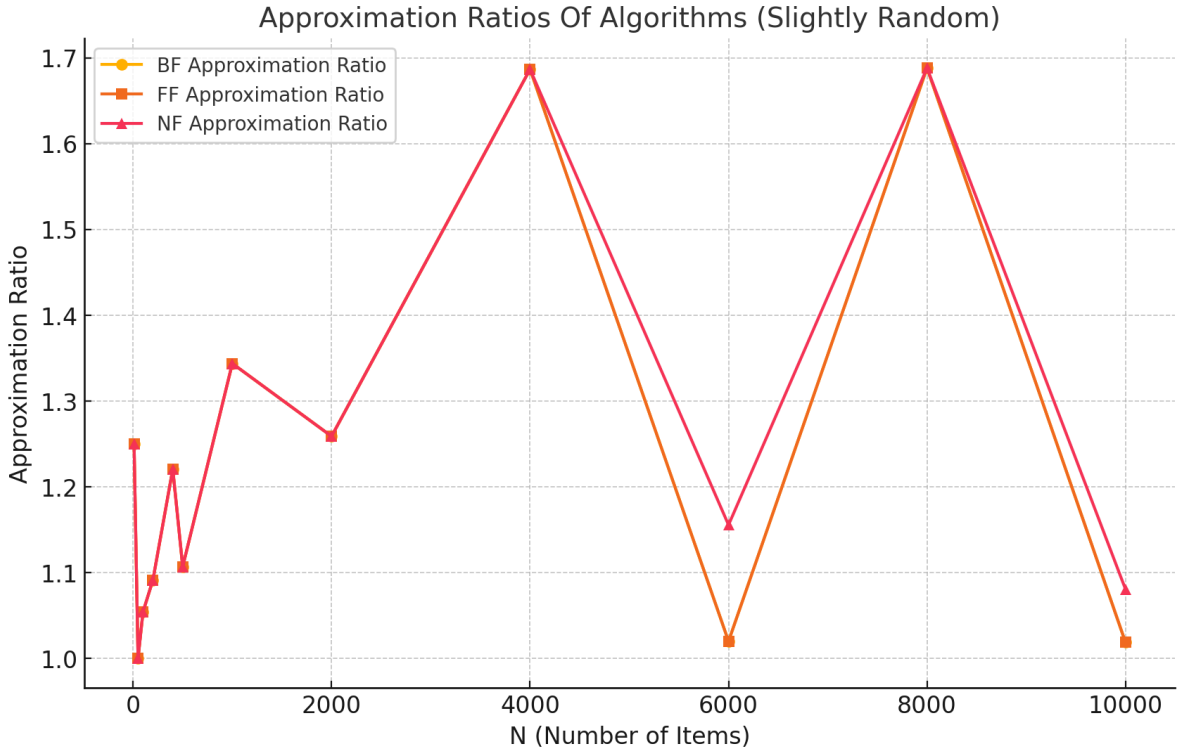
- **FF** and **BF** achieve approximation ratios close to **1.0**, indicating high solution quality.

- **NF** stabilizes at an approximation ratio of approximately **1.3**, reflecting lower solution quality compared to FF and BF.

Case 2: Slightly Random

Here, we randomly determine a base value, and both the width and height fluctuate within 10% of this base value. This means we generate N rectangles that are approximately the same size, resembling squares. The final results exhibit upward and downward fluctuations.





Observing the approximation ratios, we can understand the nature of these upward and downward fluctuations:

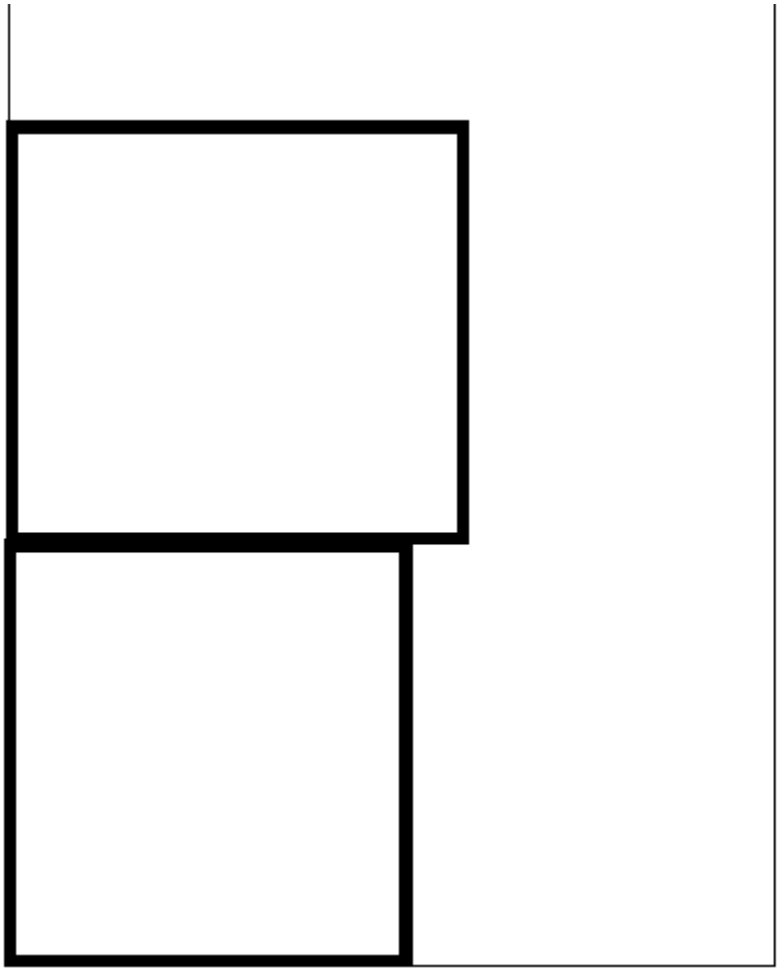
- When N is relatively small, the results exhibit high randomness due to the varying combinations of rectangles, similar to the previous case.

- When N becomes sufficiently large, the results can be divided into **two categories**:
    1. **Cases like $N = 6000$ and $N = 10000$, where the approximation ratio approaches 1.0:**
       Here, the randomly generated base values are almost divisible by W, allowing multiple squares to completely fill the strip's width. Additionally, the squares have consistent heights, meaning each layer is nearly full.
    2. **Cases like $N = 4000$ and $N = 8000$, where the approximation ratio approaches 1.7:**
       In this scenario, the base value is slightly larger than $W/2$, so each layer can only fit one square, leaving nearly half the space empty. **However**, the optimal solution in this case is maybe not as small as our ideal approximation since every squares can only place alone in one layer , so the **real** approximation ratio is not so big as **1.7**, but some smaller value near **1**.

For example, the following input demonstrates the second case:

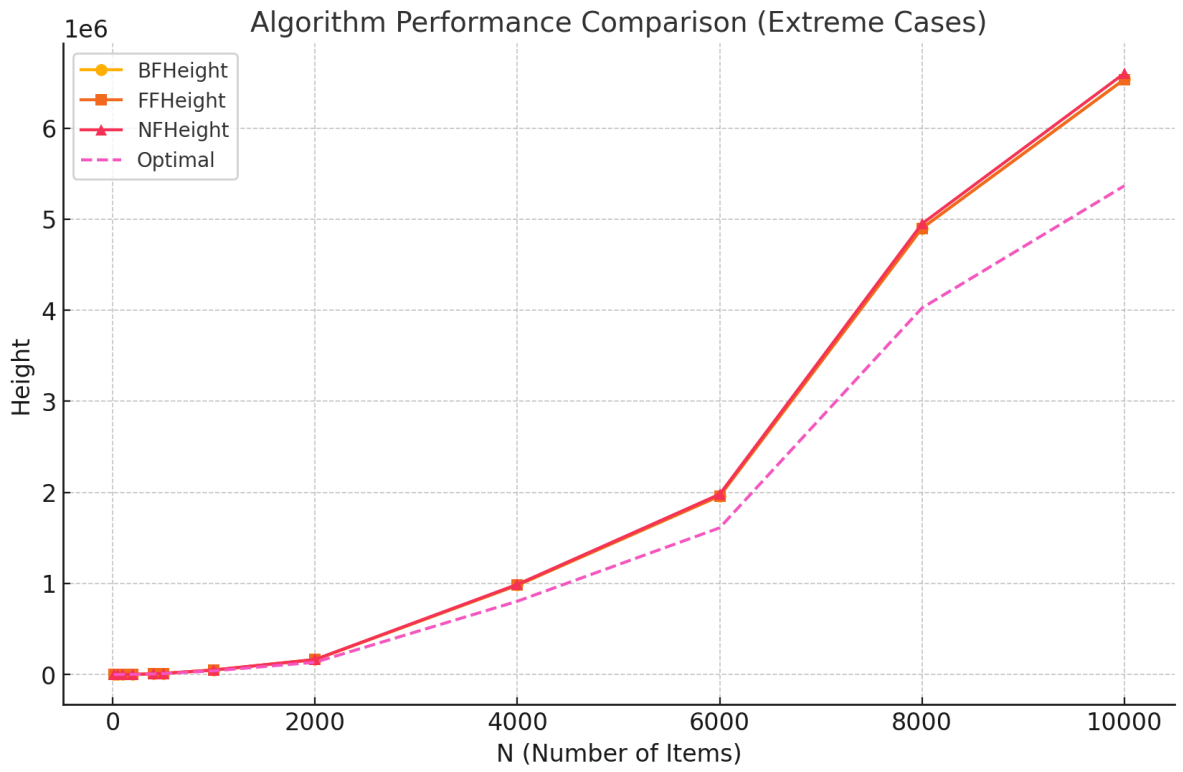
```
4000 3000
1719 1872
1811 1839
1839 1743
...
```

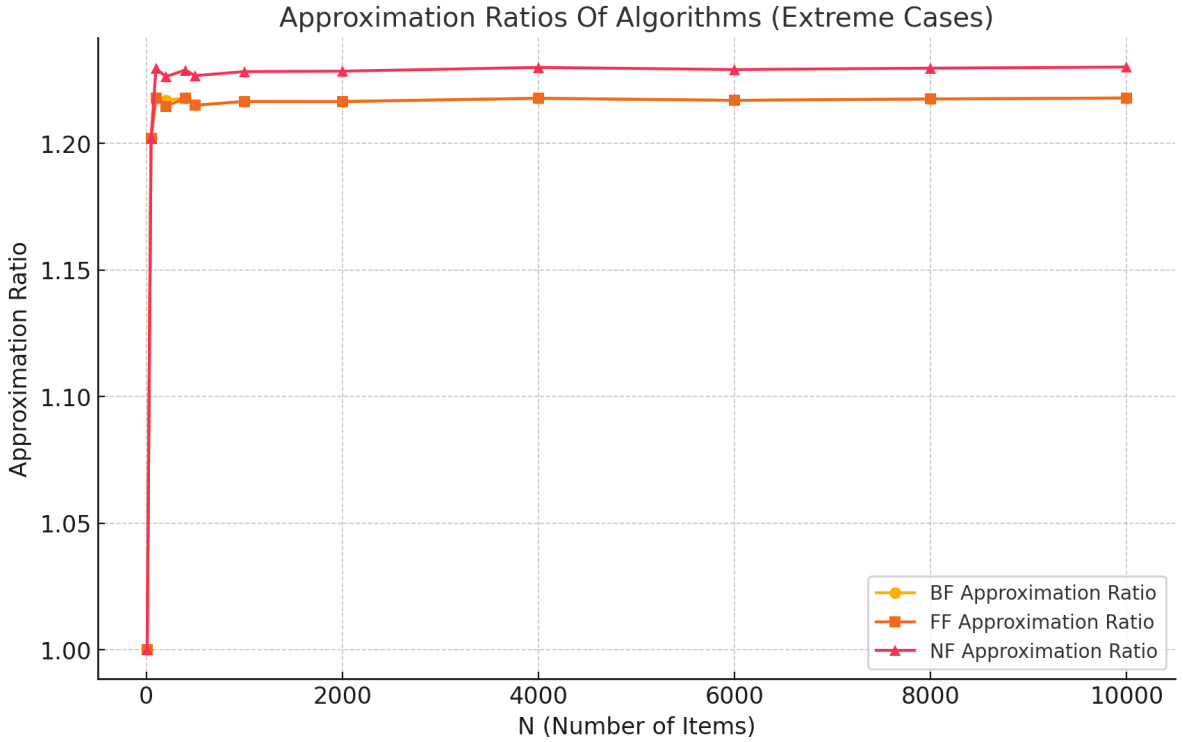
This is a good illustration of the issue where large empty spaces are left on each layer due to the sizes of the rectangles.



Case 3: Extreme Cases

Here, we define a large base value of **0.8W** and a small base value of **0.05W**. We generate **0.1N** large rectangles with dimensions fluctuating around the large base value, and **0.9N** small rectangles with dimensions fluctuating around the small base value. This case represents an extreme distribution, consisting of a few very large rectangles and many very small rectangles.





Observing the approximation ratios in this case, we find that all three algorithms stabilize around **1.22** when N is not particularly small. This indicates that this type of input distribution results in relatively poor outputs for all three algorithms.

Here, we reference images from other sources to illustrate the comparison between the suboptimal cases produced by the **FF** algorithm and the optimal solutions.

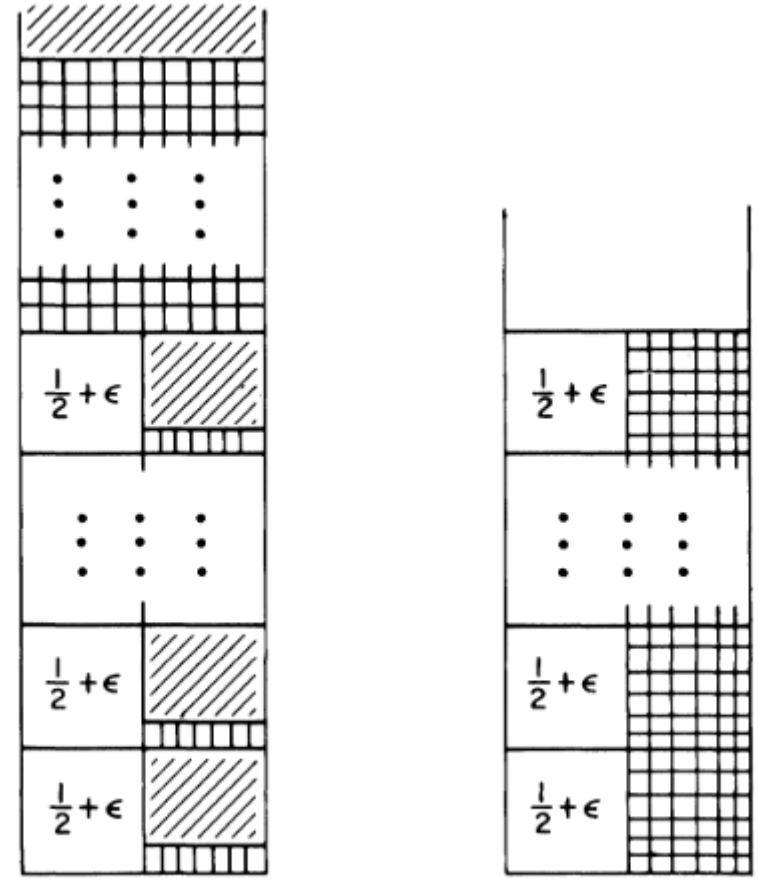


FIG. 5. *Worst-case examples for Theorem* 4.

In this image, the **FFDH** algorithm handles the input by first processing the large rectangles, placing each large rectangle on a separate layer. Afterward, it processes the small rectangles, filling the gaps in each layer horizontally. This approach wastes vertical space, and any remaining small rectangles are placed on new layers at the top.

The optimal placement method, however, would fill the gaps left by the large rectangles both horizontally and vertically with small rectangles. Using this method, the **approximation ratio** can be improved to a maximum of **1.5**.

Thorough Analysis

1. Number of Rectangles (n)

When n is small, the approximation ratio tends to fluctuate significantly. This is because the random combinations of rectangles can lead to high variability in how well they fit into the container. For example, a small number of poorly arranged rectangles might leave large gaps or require extra layers, causing the approximation ratio to deviate from optimal. As n grows, the ratio stabilizes because the law of large numbers ensures that the randomness in rectangle placement has less impact, leading to more predictable and consistent packing results.

2. Rectangle Width Relative to W

- **Widths Divisible by W**:
  If the widths of the rectangles are divisible by W, they can perfectly fill the width of the container without leaving any horizontal gaps. This results in highly efficient packing and a lower approximation ratio.
- **Widths Slightly Greater than $W/2$**:
  When the rectangle widths are slightly larger than $W/2$, only one rectangle can fit in each layer, leaving nearly half the width of the container unused. This creates significant inefficiency, leading to higher approximation ratios. Such cases are particularly problematic for algorithms like FF and BF, as they cannot optimize for this type of wasted space effectively.

3. Height Variability

The variability in rectangle heights also plays a critical role:

- **Minimal Height Differences**:
  When rectangle heights are similar, layers are more uniform, and the packing is more efficient. This minimizes wasted vertical space and results in a lower approximation ratio.
- **Extreme Height Differences**:
  If the height differences are large, the packing becomes less efficient. Tall rectangles force the creation of high layers, leaving gaps that cannot be effectively filled by much shorter rectangles. This leads to increased approximation ratios, especially in algorithms like FF and NF that do not prioritize minimizing vertical waste.

Chapter 4

In this chapter, we will analyze the complexity of the program and provide comments on its implementation.

Time complexity

First, we insert all rectangle data into a **max-heap**, which takes $O(N \log N)$ time. Then, we process each rectangle by popping it from the heap, where the pop operation costs $O(\log N)$. For each rectangle, we need to iterate through all existing layers, which takes $O(L)$ time for a single operation, where L is the current number of layers.

With N rectangles in total, this results in a time complexity of $O\left(\sum_{i=1}^{N} L_i\right)$. In the **worst case**, each rectangle occupies its own layer, meaning $L_i = i - 1$. Substituting this, we get:
$O\left(\sum_{i=1}^{N}(i-1)\right) = O\left(\frac{N(N-1)}{2}\right) = O(N^2)$.

Thus, the overall time complexity is $O(N^2)$.

The above analysis shows that the **First-Fit** and **Best-Fit** algorithms both have a time complexity of $O(N^2)$. For the **Next-Fit** algorithm, during its execution, it needs to pop an element from the max-heap for each rectangle, which takes $O(\log N)$, followed by $O(1)$ time to determine which layer to place it in. Therefore, the total running time is $O(N \log N)$.

Space complexity

We maintain a **max-heap** and its copy, both of which require $O(N)$ space. Additionally, the input rectangle data also requires $O(N)$ space. Therefore, the overall space complexity is $O(N)$.

This analysis applies to all three algorithms, as they all rely on the same data structures.

Analysis and comment

1. Time Complexity Analysis

- **First-Fit (FF)** and **Best-Fit (BF)**:
  Both algorithms exhibit a time complexity of $O(N^2)$. This is due to the need to traverse all existing layers for each rectangle to find the appropriate placement, which scales poorly with the increasing number of rectangles. While these algorithms are straightforward and effective for small problem sizes, their performance can become a bottleneck in large-scale instances.
- **Next-Fit (NF)**:
  With a time complexity of $O(N \log N)$, **Next-Fit** outperforms the other two algorithms in terms of running time. This is because it leverages the max-heap structure to efficiently determine the next rectangle placement, requiring only $O(\log N)$ per rectangle. This makes it more suitable for larger datasets where runtime efficiency is critical.

---

2. Solution Quality

1. Next-Fit Decreasing Height (NFDH)

- **Approximation Ratio**:
  The approximation ratio for **NFDH** is:
  $\mathrm{NFDH}(I) \leq 2 \cdot \mathrm{OPT}(I) + 1$
  This means that in the worst case, the solution produced by NFDH can be at most twice the optimal solution, plus a small constant. The asymptotic bound of 2 is tight, indicating that this is the best achievable bound for the algorithm.
- **Experimental Observations**:
  Experiments confirm that **NF** tends to produce lower-quality solutions compared to **FF** and **BF**, often with a higher approximation ratio. This aligns with the theoretical bound, as NF does not effectively minimize wasted space.

---

2. First-Fit Decreasing Height (FFDH)

- **Approximation Ratio**:
  The approximation ratio for **FFDH** is:
  $\mathrm{FFDH}(I) \leq \frac{17}{10} \cdot \mathrm{OPT}(I) + 1$
  This bound is tighter than that of NFDH, indicating that **FF** generally produces better solutions. The asymptotic bound of $17/10$ is tight, highlighting its relative efficiency in packing.
- **Experimental Observations**:
  Experimental results show that **FF** often produces solutions close to the theoretical optimum, with approximation ratios consistently better than those of **NF**. This is expected, as **FF** always fits rectangles into the first available layer, effectively minimizing wasted height.

3. Best-Fit Decreasing Height (BFDH)

- **Theoretical Approximation Ratio**:
  The material does not provide a theoretical bound for **BF**.
- **Experimental Observations**:
  Experimental data consistently demonstrates that **BF** produces solutions at least as good as, and often better than, those from **FF**. This is because **BF** selects the most optimal layer for placement at each step, resulting in better space utilization.

3. Overall Evaluation

- **Time Performance**:
  For scenarios where runtime efficiency is paramount, **Next-Fit** is the clear choice due to its $O(N \log N)$ complexity. However, for smaller datasets where runtime is less critical, the slower **First-Fit** and **Best-Fit** algorithms can still be viable options.
- **Solution Quality**:
  In terms of solution quality, **Best-Fit** generally outperforms the other two, producing solutions closer to the theoretical optimal. **First-Fit** follows closely behind, while **Next-Fit** may sacrifice quality for speed.

Conclusion

The choice of algorithm depends on the specific requirements of the problem. If solution quality is the priority, **Best-Fit** is the preferred choice. For a balance between solution quality and runtime, **First-Fit** is a reasonable option. However, if runtime efficiency is critical, especially for large-scale problems, **Next-Fit** is the most suitable algorithm despite its potentially lower solution quality.

Citation

The two inequalities for the theoretical approximation ratios and the illustrations in the third type of data are referenced from the following article.

> Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms
>
> E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan
>
> SIAM Journal on Computing 1980 9:4, 808-826

Appendix

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define MAX_N 10010       // Project设置数据量N最大为10000
typedef struct{           // 用于储存每个纹理
    int width;            // 宽度
    int height;           // 高度
} Item;
typedef struct{           // 用于储存每个层级level
    int height;           // 层级高度
    int rest_w;           // 剩余宽度
} Level;
```

```
Item item[MAX_N], copy[MAX_N];  // item索引值从1到N ,用于维护一个最大堆
Level level[MAX_N];             // level索引值从0到N-1，level用于储存层级
int level_num = 0;              // 层级数量
int N, W;                       // N为矩形数量，W为最大宽度
int heap_size = 0;              // 堆内元素数量

// 交换两个堆中元素
void swap(int x, int y) {       // 两个元素的索引值x,y
    Item t;
    t = item[x]; item[x] = item[y]; item[y] = t;
}

// 将一个元素最大堆  按高度从大到小排序
void heap_push(Item x) {
    Item t;
    int i, p;
    heap_size++;                // 堆容量+1
    item[heap_size] = x;        // 将元素加入至堆末位
    i = heap_size;
    while (i > 1) {             // 进行堆平衡维护
        p = i / 2;
        if (item[i].height > item[p].height) {  // 符合上滤条件
            swap(i, p);         // 上滤
        } else {
            break;
        }
        i = p;
    }
}

// 在最大堆中弹出并删除最大值  重新调整平衡
Item heap_pop() {
    Item target, t;
    target = item[1];           // 储存最大值，用于函数末输出
    swap(1, heap_size);         // 交换堆首与堆尾，即删除了最大值
    heap_size--;                // 堆容量-1
    int p = 1, lc, rc;
    while (p < heap_size) {     // p未至堆边界
        lc = p * 2;             // lc为p左子树
        rc = p * 2 + 1;         // rc为p右子树
        if (lc > heap_size) {   // 左子树溢出，结束循环
            break;
        }
        if (rc > heap_size) {   // 右子树溢出
            if (item[p].height < item[lc].height) {  // 判断该节点与左子树是否平衡
                swap(p, lc);
            }
            break;              // 结束循环
        }
        if (item[p].height > item[lc].height && item[p].height > item[rc].height)
{  // 如果目前已经平衡，结束循环
            break;
        }
        if (item[lc].height < item[rc].height) {    // 如果该节点p和左子树与右子树不平
衡
            swap(p, rc);          // 下滤至右子树
```

```
            p = rc;
        } else {
            swap(p, lc);          // 下滤至左子树
            p = lc;
        }
    }
    return target;               // 返回储存的最大值
}

// 判断堆是否为空
int IsEmpty() {
    return heap_size == 0;
}

// 每次重复执行指令时需要的初始化
void Initialize() {
    for (int j = 1; j <= N; j++) {      // 初始化堆
        item[j] = copy[j];
    }
    heap_size = N;                       // 初始化堆容量
    level_num = 0;                       // 初始化层级数量
}

// BestFit算法判断是否可塞进一个最合适的层级 (最合适：使塞入后的层级的剩余水平宽度最小)
int BestFitable(Item x) {
    int min = 1000000, min_index = -1;     // 初始化
    if (level_num == 0) {                 // 无层级直接返回-1
        return -1;
    }
    for (int i = 0; i < level_num; i++) {
        if (level[i].height >= x.height && level[i].rest_w >= x.width) {  // 判断能否塞入
            if (level[i].rest_w - x.width < min) {    // 判断塞入后该层级的剩余水平宽度是否最小
                min = level[i].rest_w - x.width;      // 更新
                min_index = i;
            }
        }
    }
    if (min_index == -1) {         // 不能塞入
        return min_index;
    } else {
        level[min_index].rest_w = level[min_index].rest_w - x.width;  // 更新被塞入的层级的宽度
        return min_index;
    }
}

// FirstFit算法会判断直接塞入第一个可塞入的层级
int FirstFitable(Item x) {
    if (level_num == 0) {            // 无层级直接返回-1
        return -1;
    }
    for (int i = 0; i < level_num; i++) {
        if (level[i].height >= x.height && level[i].rest_w >= x.width) {    // 判断是否能塞入,能塞入直接更新相关数据，退出函数
```

```
            level[i].rest_w = level[i].rest_w - x.width;
            return 1;
        }
    }
    return -1;           // 无法塞入
}

// NextFit算法判断是否能放入当前层级
int NextFitable(Item x) {
    if (level_num == 0 || level[level_num - 1].rest_w < x.width) {
        return -1;  // 如果当前层级剩余宽度不足以容纳条形图，返回-1
    } else {
        level[level_num - 1].rest_w -= x.width;  // 放入当前层级
        return 1;
    }
}

// NextFit算法
int Next_Fit_Find() {
    Item x;
    int flag;
    while (IsEmpty() != 1) {           // 堆非空则继续执行
        x = heap_pop();                 // 按高度从大到小开始
        flag = NextFitable(x);          // 判断是否能塞入
        if (flag == -1) {               // 如果无法放入，则创建新层级
            level_num++;                // 创建新的层级
            level[level_num - 1].height = x.height;
            level[level_num - 1].rest_w = W - x.width;
        }
    }
    int sum_height = 0;
    for (int i = 0; i < level_num; i++) {
        sum_height += level[i].height;
    }
    return sum_height;  // 返回总的高度
}

// BestFit算法
int Best_Fit_Find() {
    Item x;
    int flag;
    while (IsEmpty() != 1) {           // 堆非空则继续执行
        x = heap_pop();                 // 按高度从大到小开始
        flag = BestFitable(x);          // 判断是否能塞入
        if (flag == -1) {               // 无法放入
            level_num++;                // 创建新的层级
            level[level_num - 1].height = x.height;
            level[level_num - 1].rest_w = W - x.width;
        }
    }
    int sum_height = 0;
    for (int i = 0; i < level_num; i++) {
        sum_height += level[i].height;
    }
    return sum_height;  // 返回总的高度
}
```

```
// FirstFit算法
int First_Fit_Find() {
    Item x;
    int flag;
    while (IsEmpty() != 1) {            // 堆非空则继续执行
        x = heap_pop();                  // 按高度从大到小开始
        flag = FirstFitable(x);          // 判断是否能塞入
        if (flag == -1) {                // 无法放入
            level_num++;                 // 创建新的层级
            level[level_num - 1].height = x.height;
            level[level_num - 1].rest_w = W - x.width;

        }
    }
    int sum_height = 0;
    for (int i = 0; i < level_num; i++) {
        sum_height += level[i].height;
    }
    return sum_height;  // 返回总的高度
}

int main(void) {
    // 声明相关变量，打开文件读取和写入
    clock_t start, stop;
    double total_time;
    freopen("Texture_in.txt", "r", stdin);
    freopen("Texture_out.txt", "w", stdout);
    scanf("%d %d", &N, &W);    // 读取样本数量N和指定宽度W
    Item temp;
    for (int i = 0; i < N; i++) {
        scanf("%d %d", &temp.width, &temp.height);
        heap_push(temp);         // 将样本数据弹入堆中，进行堆排序
    }
    for (int i = 1; i <= N; i++) {
        copy[i] = item[i];       // 对最大堆进行备份，方便多次循环
    }

    int k = 1000;    // 循环次数,可改;由于程序执行较快，时间变化不明显，设置1000次循环执行为一组;
    start = clock();         // 开始计时
    int Best_Fit_Minimum_Height;
    for (int i = 1; i <= k; i++) {
        Initialize();         // 初始化
        Best_Fit_Minimum_Height = Best_Fit_Find();
    }
    stop = clock();          // 停止计时
    total_time = ((double)(stop - start)) / 1.0 / CLK_TCK;
    // 输出BestFit的最低高度与运行时间
    printf("Best_Fit_Minimum_Height: %d\n", Best_Fit_Minimum_Height);
    printf("Best_Fit_Find_Time: %lfs\n", total_time);

    start = clock();         // 开始计时
    int First_Fit_Minimum_Height;
    for (int i = 1; i <= k; i++) {
        Initialize();
```

```
        First_Fit_Minimum_Height = First_Fit_Find();
    }
    stop = clock();         // 停止计时
    total_time = ((double)(stop - start)) / 1.0 / CLK_TCK;
    // 输出FirstFit的最低高度与运行时间
    printf("First_Fit_Minimum_Height: %d\n", First_Fit_Minimum_Height);
    printf("First_Fit_Find_Time: %lfs\n", total_time);

    // NextFit算法
    start = clock();        // 开始计时
    int Next_Fit_Minimum_Height;
    for (int i = 1; i <= k; i++) {
        Initialize();
        Next_Fit_Minimum_Height = Next_Fit_Find();
    }
    stop = clock();         // 停止计时
    total_time = ((double)(stop - start)) / 1.0 / CLK_TCK;
    // 输出NextFit的最低高度与运行时间
    printf("Next_Fit_Minimum_Height: %d\n", Next_Fit_Minimum_Height);
    printf("Next_Fit_Find_Time: %lfs\n", total_time);
}
```