Advanced Data Structures and Algorithm Analysis

Project 6:
Texture Packing

Date: 2024-12-02

2024-2025 Autumn&Winter Semester

Table of Contents

Chapter 1: Introduction

1.1 Problem Description

The project require us to design **approximation algorithms** running in polynomial time to solve **Texture Packing** problem. We can regard it as a 2-dimension bin packing, with items("rectangle texture" in the problem) and bins("resulting texture" in the problem) having both width and height, but we only need a single bin with **bounded width** and **unbounded height**, and we should keep the bin with a (nearly) minimum height.

1.2 Backgound of Algorithms

1.2.1 BL Algorithm

BL algorithm (Bottom-up left-justified algorithm) was first described by Baker et al. It works as follows: Let L be a sequence of rectangular items. The algorithm iterates the sequence in the given order. For each considered item $r \in L$, it searches for the bottom-most position to place it and then shifts it as far to the left as possible. Hence, it places at the bottom-most left-most possible coordinate (x, y) in the strip.

1.2.2 FFDH Algorithm

Just like texture packing problem is the 2D version of bin packing problem, the **FFDH** (i.e. First-Fit Decreasing-Height) algorithm is also the 2D version of FFD algorithm in bin packing problem.

- It's an **offline algorithm**, which means that the algorithm doesn't process the input data unless it gets all input data, and in our algorithm, all items should be sorted by their height in a decreasing order.
- Before placing the current item, the algorithm scans the levels from bottom to top in the bin, then places the item in the first level where it will fit.
- A new level will be created only if the item does not fit in any previous ones.
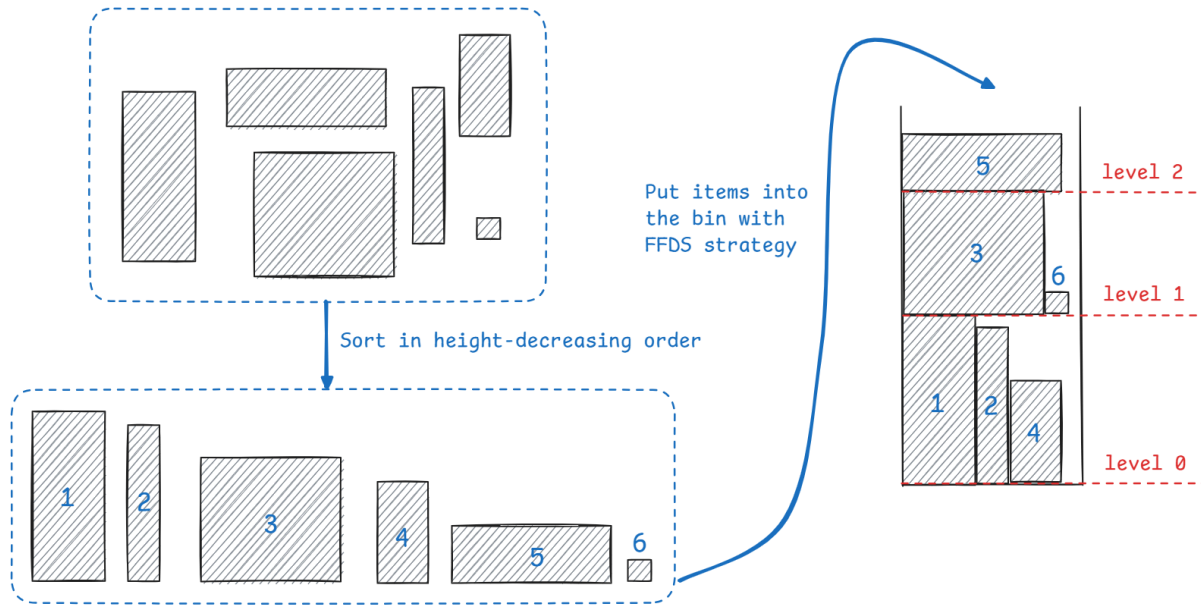
Figure 1: FFDH Approximation Algorithm

1.2.3 Mersenne Twister Algorithm

The Mersenne Twister is an efficient pseudo-random number generator known for its long period and high-quality random numbers. It's based on a specific Mersenne prime (a prime number of the form $(2^p - 1)$), using a 624-word state vector of 32-bit integers. It generates random numbers through a series of bit operations like shifts and masks. Widely used for its speed, long period, and randomness quality.

Chapter 2: Algorithm Specification

In this chapter, we will introduce approximation algorithms of FFDH(basic version) and (advanced version) in details, including aspects below, to enable readers have a comprehensive and thorough understanding of these algorithm.

- Description of approximation algotithms with pseudocodes.
- Calculation of approximation ratio of algorithm with proof.

2.1 BL Algorithm

First of all, we provide the pseudocode of BL algorithm below.

Procedure:BL(*rect*: **POINT array**, *W*: **double**, *n*: **integer**, *isDebug*: **bool**, *debugFile*: **string**)

1 **Begin**
2   Initialize upBound list with rightmost **and** initial points
3   Set *maxHeight* **to** 0
4   Sort *recs* by width **in** descending order
5   **while** there is *rec* remained **do**
6   Find the proper position of the rec
7   Place the rectangle **and** update upBound
8   Update *maxHeight* **if** needed
9   **If** *isDebug*, log upBound
10   **end**
11   **If** *isDebug*, log all *recs*
12   **return** *maxHeight*
13 **End**

Among them, upBound is maintained through a **bidirectional linked list**. The implementation of the bidirectional linked list is to directly call the `list` in the STL container.The reason for using a bidirectional linked list is that we need to frequently insert and delete upBound, but the need for search operations is very small.

Each element is a `POINT` structure as shown below.

```cpp
struct POINT{
    double x;
    double y;
    double width;

    //Overload the equal operator
    bool operator==(const POINT& other)
    {
        return x==other.x && y==other.y && width==other.width;
    };
};
```

we determine a line segment by determining the left endpoint coordinates and the length of the line segment. we record all the line segments that can be covered by the projection from top to bottom in order from right to left.

For example, for the following situation:
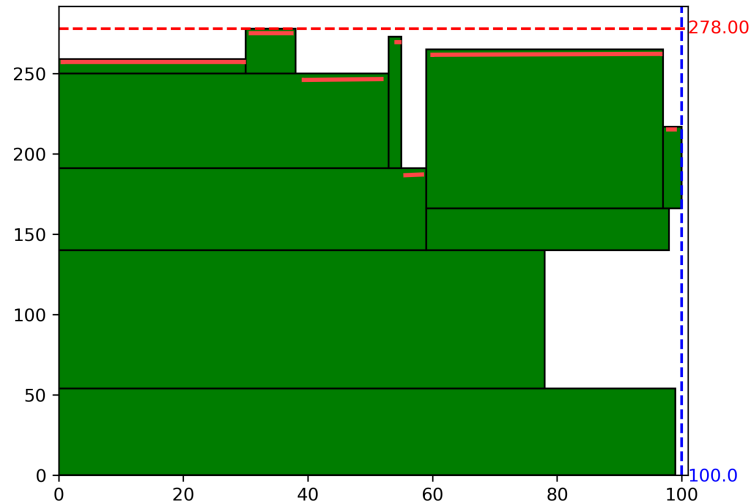
Figure 2: An Example for Records of Line Segments

The upBound will be describe as below:

```
x: 100 y: 0 width: 0        #Acting as a sentinel without affecting
the results
x: 97 y: 217 width: 3
x: 59 y: 265 width: 38
x: 55 y: 191 width: 4
x: 53 y: 273 width: 2
x: 38 y: 250 width: 15
x: 30 y: 278 width: 8
x: 0 y: 259 width: 30
```

2.1.1 Algorithm Analysis

This algorithm is a 3-approximation algorithm after sorting the rectangles in descending order of width. Here is the proof:

7

Proof

Let h^* denote the the height of the lower edge of a tallest piece whose upper edge is at height h_{BL}. If y denote the height of this piece, then $h_{\mathrm{BL}} = h^* + y$. Let A denote the region of the bin up to height h^*.

Suppose A is at least half occupied. Then we have $h_{\mathrm{OPT}} \geq \max\left(y, \frac{h^*}{2}\right)$; hence, $y > \frac{h^*}{2}$ implies:

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}} \leq \frac{y + h^*}{y} < \frac{y + 2y}{y} = 3$$

and if $y \leq \frac{h^*}{2}$, we have:

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}} \leq \frac{\frac{h^*}{2} + h^*}{\frac{h^*}{2}} = 3$$

As a result, we only need to show that A is at least half occupied.

We assert that for every horizontal line in area A, the sum of the lengths of the lines crossing the rectangle must be no less than the sum of the lengths of the lines crossing the blank area. (We can ignore all lines that cross the intersection of the rectangles, since obviously they have measure 0)

This conclusion is guaranteed by the following facts:

1. Each horizontal line must pass through a rectangle attached to the left edge. This is obvious because the width of the placed rectangle is decreasing.
2. Assuming that the horizontal line l passes through a blank area S, it is guaranteed by 1 that there must be a rectangle on its left. Secondly, it is obvious that the width of the rectangle on the left is greater than the width of S, which is guaranteed by the fact that the width of the rectangle placed in the upper layer must be greater than S (if not, S can be placed in a new rectangle, and the width of the upper rectangle is not greater than the rectangle on the left of S)

Therefore, for each horizontal line l in A, it must meet our assertion. Integrating each line, it can be seen that half of the area in A is occupied.

2.1.2 BL_Change: A Failed Attempt

We try to optimize the order of rectangles. Divide all rectangles into two groups, one with a width less than w/2, and sort them in descending order of height/width; the other with a width greater than w/2, and sort them in descending order of width.

This optimization order is based on the following observations: In the original algorithm, rectangles larger than w/2 are almost all arranged in a single row, resulting in a large waste of space on the right side.

But we failed to do better in my implementation. This optimization algorithm only has a slight advantage in small and medium-sized data. When the data volume is large enough, it will cause greater space waste due to the defects of my own algorithm implementation.

2.2 FFDH Algorithm

The main data structure we used is a `struct` called `Item`. Each `Item` contains four elements(all double values): `width`, `height`, `x` and `y`, which represent the width, the height and the lower left point of the item respectively.
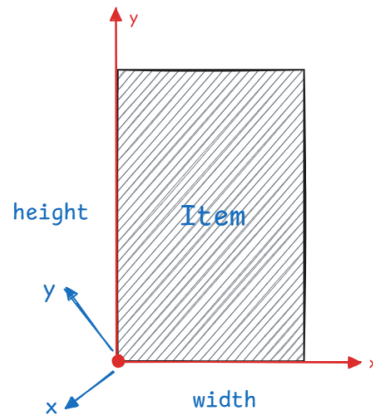

Figure 3: The `struct Item`

Now let's take a look at the pseudocode of FFDH to gain a deeper insight into this kind of approximation algorithm.

> **Inputs**:
> - *W*: Fixed width of the bin(i.e. resulting texture)
> - *n*: The number of items
> - *rect*: Multiple rectangle texture, i.e. items

- *isDebug*: Flag of debug mode
- *outFile*: Flag of file output mode

Outputs:

- *curHeight*: the "minimum" height of the bin

Procedure: FFDH(*W*: double, *n*: integer, *rect*: Item array, *isDebug*: bool, *outFile*: bool)

1 **Begin**
2   Sort *rect*[] by item's height **in** decreasing order
3   **for** *item* **in** *rect*[] **do**
4   **for** *level* **in** existing levels **do**
5   **if** *level*'s width + *item*[i]$\rightarrow$*width* $\leq$ *W* **then**
6   put the *item* into this level **and** update the state
7   **break**
8   **end**
9   **if** no level can fit the *item* **then**
10   create a new level
11   put the *item* into this level
12   update the state
13   **end**
14   **end**
15   print debug info **if** the user using the debug mode
16   **return** the current height of the bin as the "minimum" height
17 **End**

We can divide the procedure into four steps:

1. **Sort** all items by their height in decreasing order.
2. For all items, put them into the bin in the sorted order.
   - Scan all levels from bottom to top, **find the first level that can accomodate the current item**.
   - If no levels can fit it, then **create a new level** and put it into the new level.
3. (if necessary)**Print the debug info**, including:
   - the height-decreasingly sorted item data,
   - the occupied-by-items width for each level,
   - the positions of items.

4. **Return** the current height of the bin as the "minimum" height.

Now we should figure out the approximation ratio of this algorithm. It was proved that FFDH algorithm is a **2.7-approximation algorithm** (the conclusion is given by Wikipedia). Because it is difficult for us to prove this approximation ratio based on our mathematical knowledge, and the relevant proof content cannot be directly checked on the Internet (need to pay to unlock the paper), so unfortunately the proof part is omitted.

2.3 Large-scale Random Number Generation Strategy

Here we only take the generation of uniformly distributed integers as an example. The codes for other types are similar. The following code only shows the most basic operations. With these distributed numbers, generating samples is easy.

```cpp
#include<random>
using namespace std;

...

    random_device seed;
    mt19937 gen(seed());

    uniform_int_distribution<> height(1, maxHeight);
    h = height(gen);
...
```

The basic process is to call `random_device` to generate a random number seed, call `mt19937` to generate a relatively high-quality pseudo-random number, and call the corresponding random distribution class to convert the pseudo-random number into the random variable we need.

2.4 Optimal Solution Sample Generation Strategy

Procedure: square_generate(*Height*: **double**, *Width*: **double**, *hNum*: **integer**, *wNum*: **integer**)

1 **Begin**
2   Initialize *recs* as an empty vector
3   Initialize random number generator *gen*

```
 4        Initialize uniform distributions height and width
 5
 6        Initialize vectors hp and wp
 7        Add 0 and Height to hp
 8        Add 0 and Width to wp
 9
10        while (the size of hp is less than hNum + 1) do
11            Generate a random number h
12            If h is not in hp then
13                Add h to hp
14            end
15        end
16
17        while (the size of wp is less than wNum + 1) do
18            Generate a random number w
19            If w is not in wp then
20                Add w to wp
21            end
22        end
23
24        Sort hp and wp
25
26        for i from 1 to hNum do
27            for j from 1 to wNum do
28                Calculate the width and height of the rectangle
29                Add the rectangle to recs
30            end
31        end
32
33        return recs
34  End
```

As shown in the pseudo code, the large rectangle is divided into $hNum \cdot wNum$ blocks by generating split points on the edges of the large rectangle.

Chapter 3: Testing Results

In this chapter, we will test our approximation algorithms to check their correctness and performance, which lays a solid foundation for our following analysis on time complexity. We will use test tables and curve diagrams to make our explanation more graphically and vividly.

3.1 BL Algorithm

3.1.1 Correctness Tests

How to verify the correctness of an approximate algorithm? We think that it is only necessary to ensure that the approximate algorithm runs as expected and does not violate the requirements.

The following three examples test three typical situations:

- The first group is a simple case of a full square, which is easy to judge whether the logic is correct.
- The second group is a more general rectangle, which can be used to see whether the more complex position relationship is correct.
- The third group is a special case where there is an optimal solution, in which the sizes of the rectangles are close and there are no extreme rectangles. It can be seen that in the newer version.

Sample 1

- Input: 10 squares with a side length of 6 and the width is 20
- Output figure:
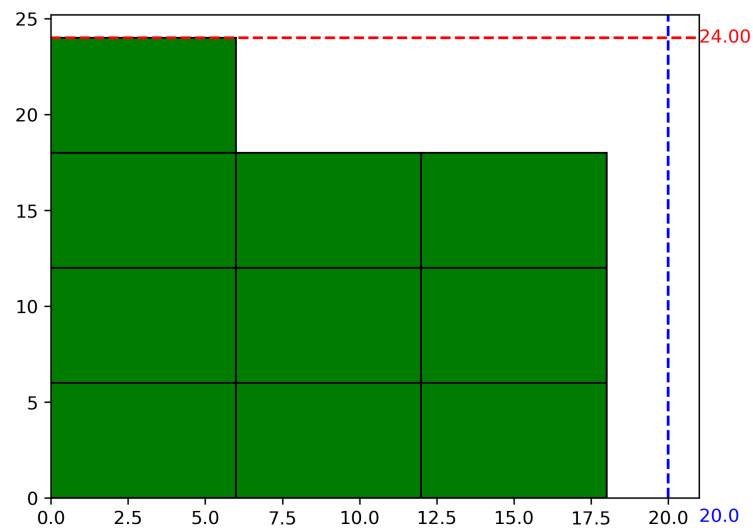
Figure 4: Correctness Test 1 for BL Algorithm

- Comment: The result is as expected.

Sample 2

- Input: 10 randomly generated rectangles and the width is 100

```
100 10
99 54
2 82
8 28
38 99
59 51
78 86
30 9
3 51
39 26
53 59
```

- Output figure:
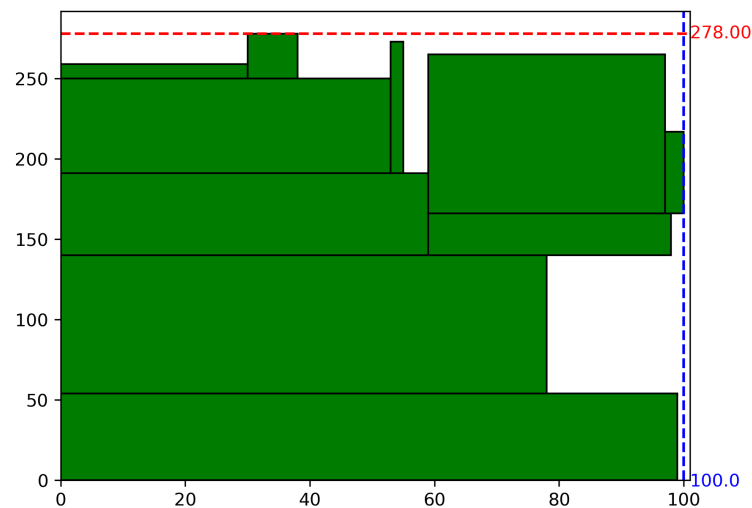
Figure 5: Correctness Test 2 for BL Algorithm

- Comment: The result is as expected.

Sample 3

- Input: This is an example of random rectangles generated by the second type of random rectangles introduced above. The size of the large rectangle is 400×200, and a total of `4*4` rectangles are generated.

```
200 16
86 272
5 272
45 272
64 272
86 72
5 72
45 72
64 72
86 13
5 13
45 13
64 13
86 43
5 43
45 43
64 43
```

- Output figure:
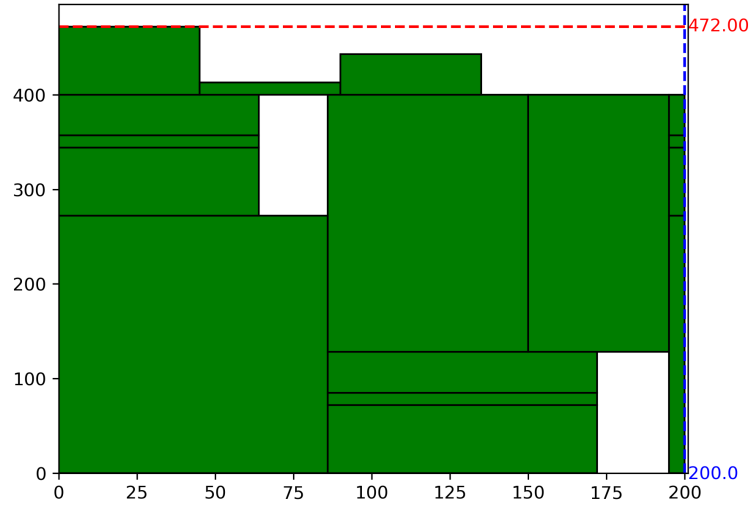
Figure 6: Correctness Test 3 for BL Algorithm

- Comment: The result is as expected.

3.1.2 Performance Tests

In order to simulate the real scene as realistically as possible, we selected samples generated by uniform distribution and normal distribution for time testing.

Since there is no essential difference between the floating point case and the integer case, only the integer case is analyzed here.

Uniform distribution

File name	Size	Time(s)
uniform_int_10	10	0.00029159
uniform_int_50	50	0.000112231
uniform_int_100	100	0.0004264
uniform_int_500	500	0.001882206
uniform_int_1000	1000	0.006107088
uniform_int_5000	5000	0.0226485
uniform_int_10000	10000	0.0588621

Table 1: Performance Tests for BL Algorithm in Uniform Distribution

The curve is drawn as follows:

16
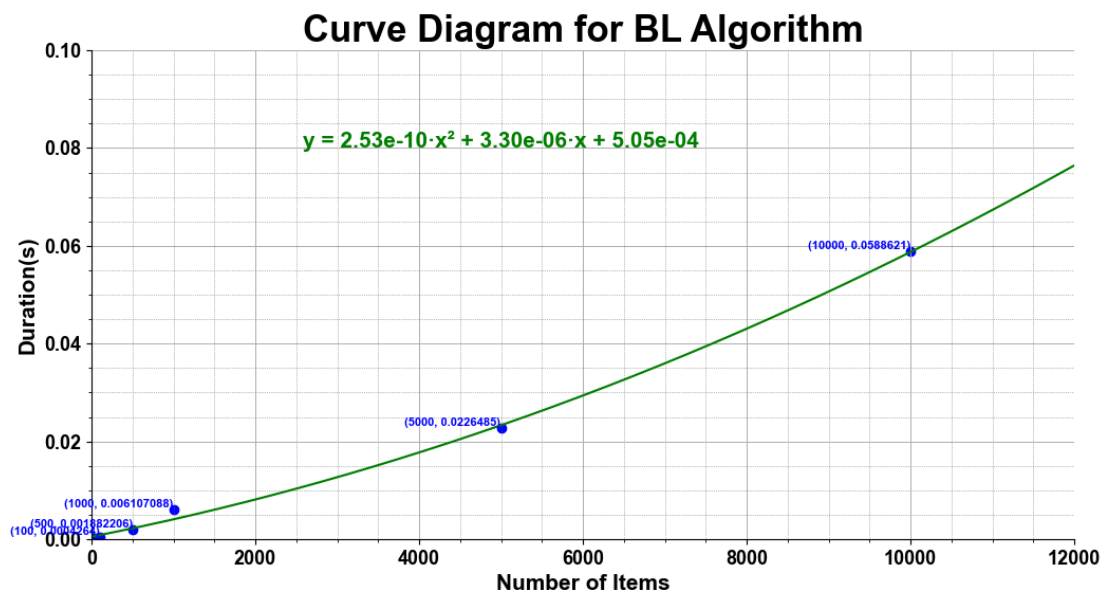
Figure 7: Curve Diagram for BL Algorithm in Uniform Distribution

Normal distribution

File name	Size	Time(s)
normal_int_10	10	0.000039636
normal_int_50	50	0.000127671
normal_int_100	100	0.00043265
normal_int_500	500	0.001962675
normal_int_1000	1000	0.00571202
normal_int_5000	5000	0.02116575
normal_int_10000	10000	0.05734626

Table 2: Performance Tests for BL Algorithm in Normal Distribution

The curve is drawn as follows:
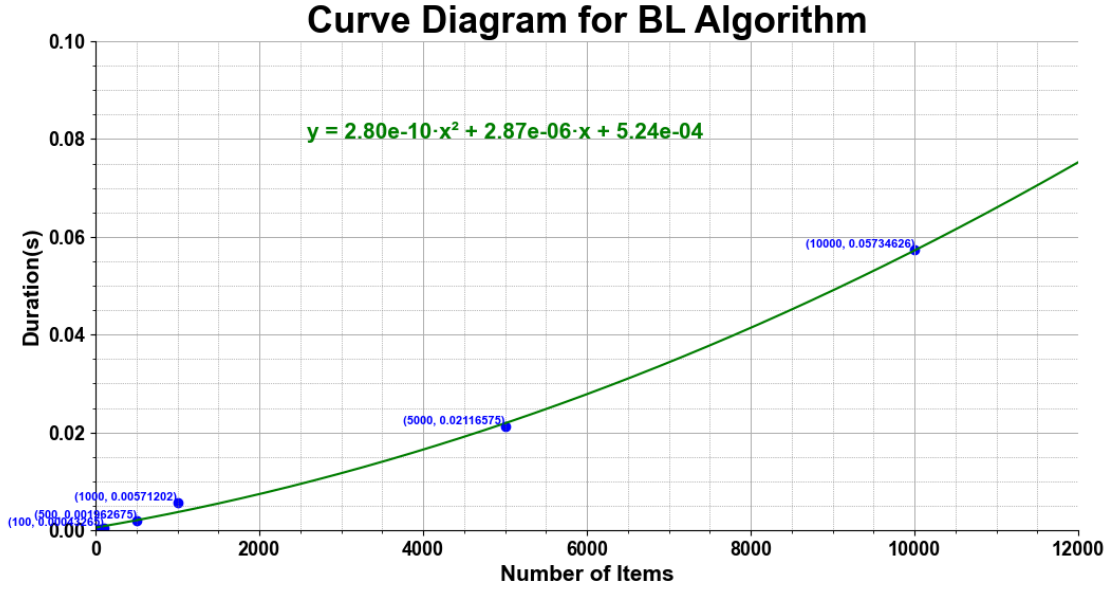
Figure 8: Curve Diagram for BL Algorithm in Normal Distribution

Analysis:

From the time complexity analysis in chapter 4, we can see that the time complexity of the BL algorithm is $O(n^2)$. However, in actual measurement, the secondary factor is not significant. The main reason may be that it is difficult for `upBound` to maintain an average size of $O(n)$ when the data scale is small, making the primary factor more prominent.

3.2 FFDH Algorithm

3.2.1 Correctness Tests

Sample 1

- Purpose: check the correctness in **the normal case, with relatively small waste space**.
- Input: See the input file in directory `./code/inputs/FFDH/inputs/input1`
- Expected Result:

Figure 9: Correctness Test 1 for FFDH Algorithm

- Actual Result:

```
./code/outputs/FFDH/outputs/output1
Debug Info:
Height-decreasingly sorted item data:
0: 6.00, 8.00
1: 5.00, 7.00
2: 4.00, 6.00
3: 7.00, 5.00
4: 3.00, 4.00
5: 4.00, 3.00
6: 6.00, 2.00

Total level: 2
Width:
0 level: 18.00
1 level: 17.00

Position:
Item 1: level 0, x: 0.00, y: 0.00
Item 2: level 0, x: 6.00, y: 0.00
Item 3: level 0, x: 11.00, y: 0.00
Item 4: level 1, x: 0.00, y: 8.00
Item 5: level 0, x: 15.00, y: 0.00
Item 6: level 1, x: 7.00, y: 8.00
Item 7: level 1, x: 11.00, y: 8.00
====================================
```

```
The total height: 35.00
The ideal height(Area / Width): 9.89
The "minimum" height: 13.00
```

As the debug info show above, our program can figure out this case properly.

Sample 2

- Purpose: check the correctness in **the normal case, with relatively large waste space**.
- Input: See the input file in directory `./code/inputs/FFDH/inputs/input2`
- Expected Result:

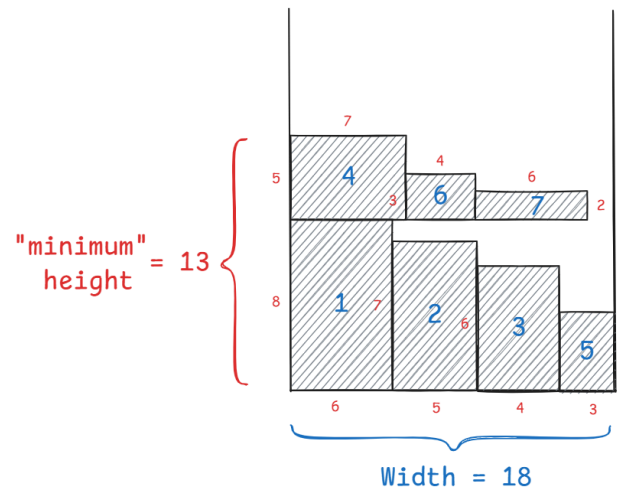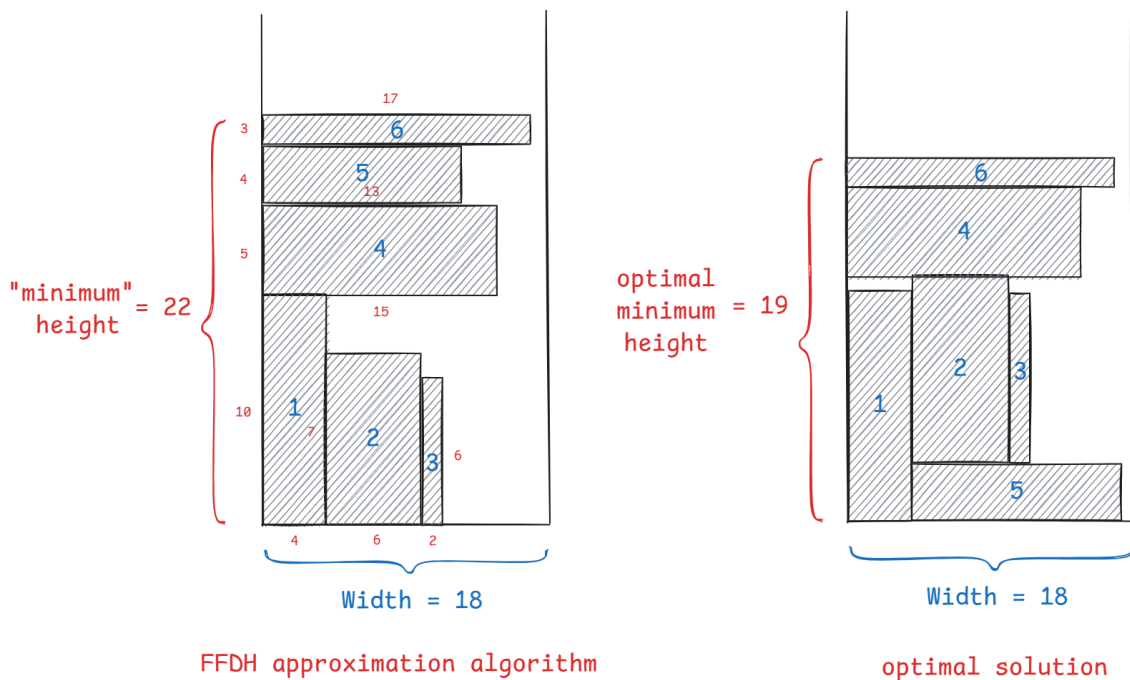Figure 10: Correctness Test 2 for FFDH Algorithm

- Actual Result:

```
./code/outputs/FFDH/outputs/output2
Debug Info:
Height-decreasingly sorted item data:
0: 4.00, 10.00
1: 6.00, 7.00
2: 2.00, 6.00
3: 15.00, 5.00
4: 13.00, 4.00
5: 17.00, 3.00
```

```
Total level: 4
Width:
0 level: 12.00
1 level: 15.00
2 level: 13.00
3 level: 17.00

Position:
Item 1: level 0, x: 0.00, y: 0.00
Item 2: level 0, x: 4.00, y: 0.00
Item 3: level 0, x: 10.00, y: 0.00
Item 4: level 1, x: 0.00, y: 10.00
Item 5: level 2, x: 0.00, y: 15.00
Item 6: level 3, x: 0.00, y: 19.00
====================================
The total height: 35.00
The ideal height(Area / Width): 15.11
The "minimum" height: 22.00
```

As the debug info show above, our program can figure out this case properly. However, there is some bias for the calculation of the algorithm calculated and the optimal solution.

Sample 3

- Purpose: check the correctness in **the large-scale case, with very large waste space**.
- Input: See the input file in directory `./code/inputs/FFDH/inputs/input3`
- Expected Result:
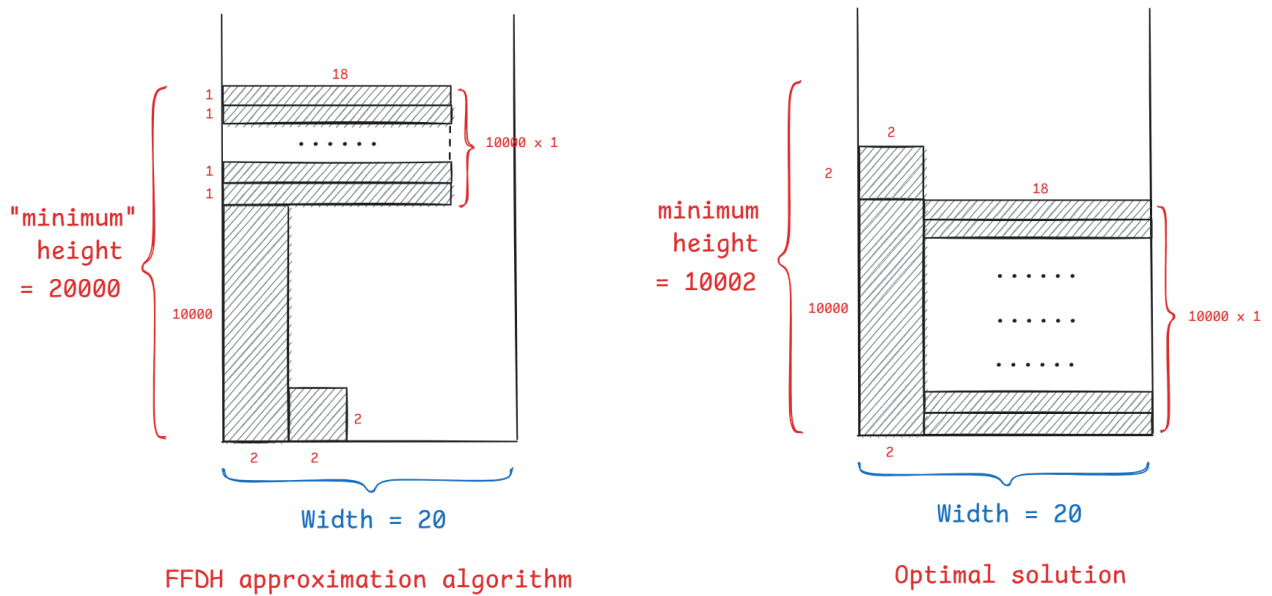
Figure 11: Correctness Test 3 for FFDH Algorithm

- Actual Result:

```
./code/outputs/FFDH/outputs/output3
Debug Info:
Height-decreasingly sorted item data:
0: 2.00, 10000.00
1: 2.00, 2.00
2: 18.00, 1.00
3: 18.00, 1.00
 ...
10000: 18.00, 1.00
10001: 18.00, 1.00

Total level: 10001
Width:
0 level: 4.00
1 level: 18.00
2 level: 18.00
3 level: 18.00
 ...
9999 level: 18.00
10000 level: 18.00

Position:
Item 0: level 0, x: 0.00, y: 0.00
```

```
Item 1: level 0, x: 2.00, y: 0.00
Item 2: level 1, x: 0.00, y: 10000.00
Item 3: level 2, x: 0.00, y: 10001.00
...
Item 10000: level 9999, x: 0.00, y: 19998.00
Item 10001: level 10000, x: 0.00, y: 19999.00
=====================================
The total height: 20002.00
The ideal height(Area / Width): 10000.20
The "minimum" height: 20000.00
```

As the debug info show above, our program can figure out this case properly. However, there is very big bias for the calculation of the algorithm calculated and the optimal solution, and $\frac{FFGH(L)}{OPT(L)} \approx 2 \le 2.7$, where $FFDH(L)$ and $OPT(L)$ are the solution of FFFH algorithm and optimal solution respectively.

3.2.2 Performance Tests

The dominant factor having infulence on the time complexity in FFDH algorithm is **the number of items**, i.e. the **input size**. Consequently, we will show the correlation between run time and input sizes by running our program in distinct input sizes, which are listed in the test table with corresponding results below.

Note that our randomly-generated input data complies the <u>uniform distribution</u>, which means that all numbers within the specified range will be selected for equal possible(this description is not mathematically rigor, because the probability of choosing one single number is zero).

Number of Items	10,000	20,000	40,000	80,000	160,000
Iterations	100	50	10	5	1
Ticks	4068	7560	6113	12382	10304
Total Time(s)	4.068	7.560	6.113	12.382	10.304
Duration(s)	0.04068	0.1512	0.6113	2.4764	10.304

Table 3: Performance Tests for FFDH Algorithm

Based on the table above, we use a Python program to draw the curve diagram of run time–input size, representing the time complexity of FFDH algorithm in a graphic and direct way.
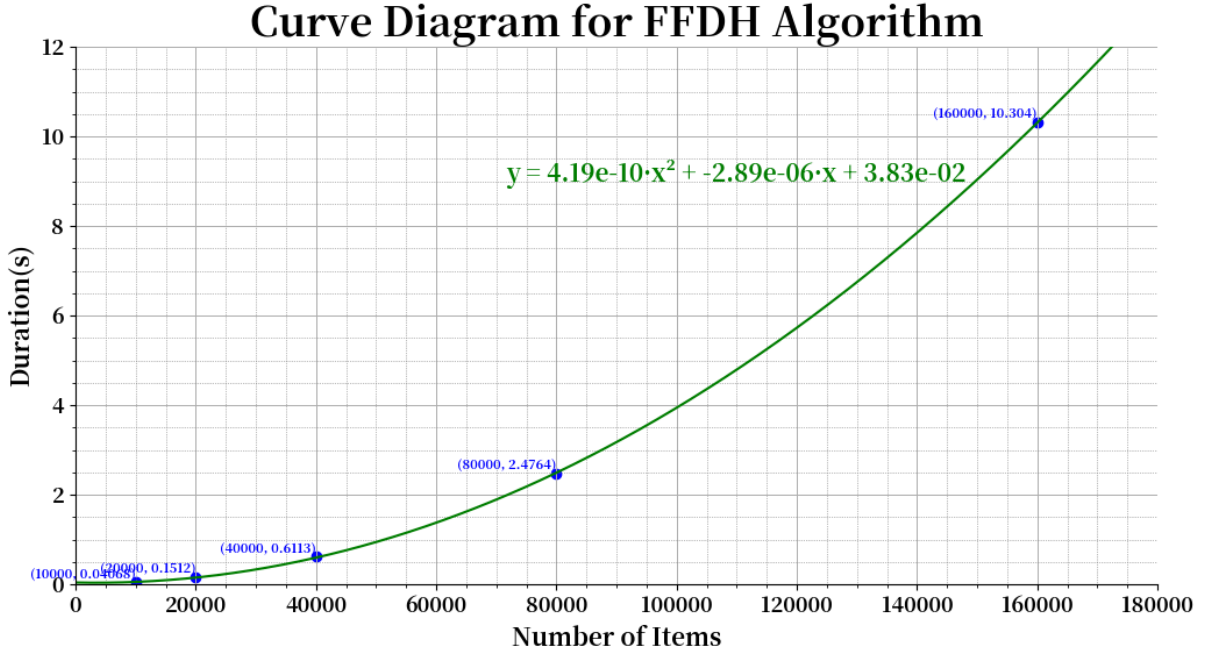
Figure 12: Curve Diagram for FFDH Algorithm in Performance Test

We use a **quadratic polynomial curve** fitting data point, and find that there is such a curve, $y = 4.19 \times 10^{-10}x^2 + 2.89 \times 10^{-6}x + 3.83 \times 10^{-2}$, that can pass almost all the data points, which shows that the FFDH algorithm can complete the calculation within the quadratic polynomial time. The theoretical analysis of time complexity will be explained in Chapter 4.

3.3 Comparative Analysis

3.3.1 FFDH Advantages analysis

Take the result graph generated by uniform_int_100 test data as an example.
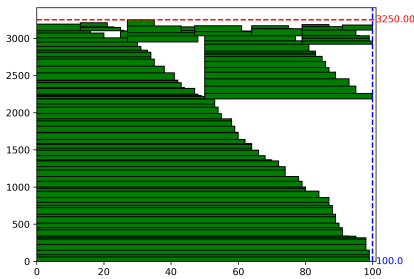
Figure 13: Ui_100_BL.png
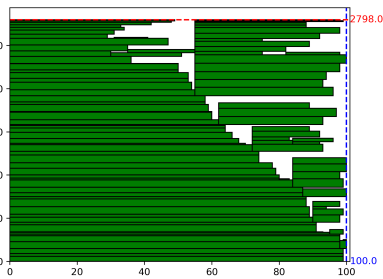
Figure 14: Ui_100_C.png
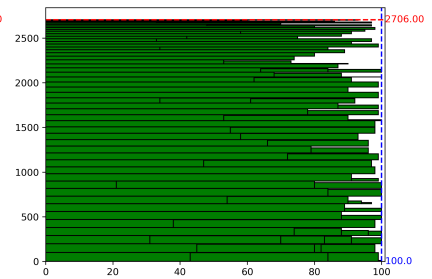
Figure 15: Ui_100_F.png

It can be seen that the FFDH algorithm has a higher space utilization rate. The main reason is that BL and even the failed improved algorithms of BL cannot solve the huge space waste caused by sorting by width. For example, in the BL algorithm, after sorting by width, all rectangles with a width greater

24

than w/2 have to be arranged in a separate row, and the degree of space waste on the other side is much greater than the degree of space waste in the vertical direction of the FFDH algorithm. This makes the FFDH algorithm obtain better results in almost all our test samples.

3.3.2 Failure of algorithm improvement

Here is a more detailed explanation of the failure of the BL_change algorithm. Take the generated image of the normal_int_10000 test data as an example.
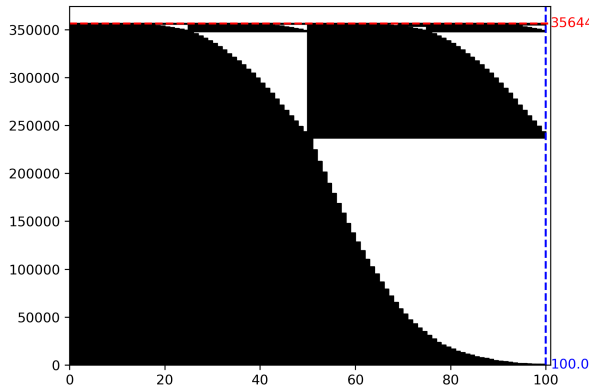
Figure 16: Ni_100_BL.png
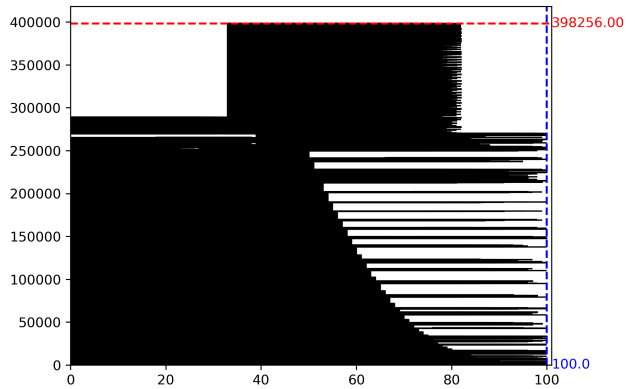
Figure 17: Ni_100_C.png

There is a very obvious anomaly in the upper half of the image of the improved algorithm. The reason is that I did not fully implement the BL algorithm to put the rectangle as far to the left as possible (my implementation method was to put the rectangle at the lowest position that can be placed, and then take the leftmost position node). However, due to time constraints, we were unable to fix this problem.

Chapter 4: Analysis and Comments

4.1 Space Complexity

Conclusion:

Both BL algorithm and FFDH algorithm are $O(N)$, where N is the number of items.

Analysis:

- BL algorithm:
  - Storing rectangle information:

    – Rectangle information is stored in `vector<rectangle> *recs`, with a space complexity of $O(N)$.

  ‣ Store upper bound information:

    – The upBound list stores the upper bound information, which contains n elements in the worst case, and has a space complexity of $O(N)$.

  ‣ Temporary variables and auxiliary data structures:

    – vector kill is used to store the points to be deleted, which contains n elements in the worst case, and has a space complexity of $O(N)$.

    – Other temporary variables (such as height, rightPoint, pointCase, etc.) occupy constant space, and have a space complexity of $O(1)$.

Based on the above analysis, the space complexity of the entire algorithm is $O(N)$.

- **FFDH algorithm**: Except the single variables, we have used some arrays, including `rect[]`, `curWidth[]` and `pos[]`, which contain the infomation of items, current width for each level and the position of each item respectively. Apparently, the level is less than or equal to the number of items(we use N to represent it). As a consequence, these three arrays are proportional to N, and the total space is less than $c \cdot N$, when c is just a constant.

4.2 Time Complexity

Conclusion:

Both BL algorithm and FFDH algorithm are $O(N^2)$, where$N$ is the number of items.

Analysis:

- BL algorithm:

  ‣ Initialization and sorting:

    – `sort`: The time complexity of the sorting operation is $O(N \log N)$.

  ‣ Main loop:

    – The main loop `while (cnt < (*recs).size())` runs n times.

    – In each loop, the inner loop `for (auto p=upBound.begin(); p! =upBound.end(); p++)` traverses the upBound list. The length of the upBound list can reach n in the worst case, so the time complexity of the inner loop is $O(N)$.

- – In the inner loop, there is also a nested loop `for (auto it=p; it!=upBound.begin(); it--)`, which also needs to traverse the upBound list in the worst case, with a time complexity of $O(N)$.
  - – Overall, the time complexity of the main loop is $O(N^2)$.
  - ‣ **Insert and delete operations**:
    - – Operations such as `upBound.insert(p, {...})` and `upBound.remove(*p)` require $O(N)$ time in the worst case, because upBound is a linked list.

Based on the above analysis, the time complexity of the entire algorithm is $O(N^2)$.

- • **FFDH algorithm**:
  - ‣ Before putting all items into the bin serially, we use the built-in function `qsort()`(i.e. the quick sort) to sort these items by their height in decreasing order. As we known in FDS course, the average time complexity of quicksort is $O(N \log N)$, and the worst time complexity is $O(N^2)$.
  - ‣ Now let's consider the core part of FFDH algorithm: it consists of a loop with two layers, the outer one corresponds to N directly, the inner one is controlled by `level`. As we have analyzed in Space Complexity, `level` $\leq N$. So the overall time consumption of the loop is less than cN^2, when c is a constant.
  - ‣ The last part of the algorithm is printing the debug info. Since it just prints the information of items sequentially, its time complexity is just $O(N)$.
  - ‣ In a nutshell, the total time complexity is $O(N^2)$.

4.3 Further Improvement

1. **Algorithm Refinement**:
   - • Use a modified version of the RF algorithm. First, put rectangles with a width greater than w/2 at the bottom in order. Then sort rectangles with a width less than w/2 by height. Take a layer from the remaining rectangles each time (similar to the FFDH algorithm), and then alternately descend (similar to BL, but once as close to the lower right as possible, once as close to the lower left as possible, and alternately).

- Since the RF algorithm is a 2-approximation algorithm, this algorithm can probably also be close to 2-approximation.

2. **Testing sample construction**: Although we come up with some samples for correctness tests, probably some crucial tests are still lost, and potential bugs may exists in our programs owing to our incomplete consideration. From our standpoint, it's difficult to find all cases for a program, but we're fully convinced that by delicate techniques and tricks for testing results, we can come up with tests as complete as possible.

3. **Original algorithm design**: Since there are many approximation algorithms that solve such problems nowadays, and our ability and time are limited, the two algorithms we implemented are actually based on existing algorithms (but we only refer to the algorithm theory, the code is our originality). In the future, with the comprehensive understanding of the approximity algorithm, we believe that we have the ability to design a completely original approximate algorithm.

Appendix: Source code

5.1 File Structure

```
.
└── code
    ├── README.md
    ├── build
    │   └── Makefile
    ├── fig
    ├── inputs
    │   ├── BL
    │   └── FFDH
    ├── outputs
    │   ├── BL
    │   └── FFDH
    └── scripts
        ├── headers
        │   ├── header.h
        │   └── ttpHeader.h
        ├── BL.cpp
```
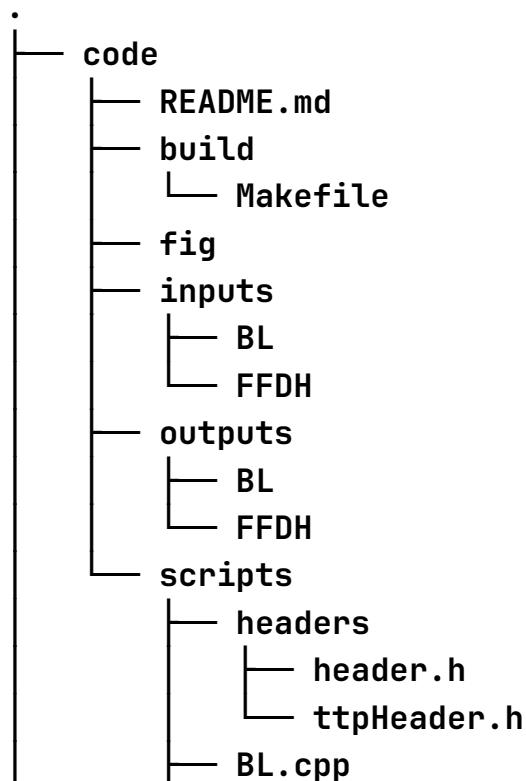
```
        ├── BLMain.cpp
        ├── generate.cpp
        ├── generate_main.cpp
        ├── ttpMain.cpp
        ├── FFDH.cpp
        ├── curve.py
        └── draw.py
    └── documents
        └── report.pdf
```

5.2 header.h

```cpp
#ifndef _HEARDER_
#define _HEARDER_ 0

#include<iostream>      // std::cout, std::cin
#include<cstdbool>      // bool
#include<vector>        // std::vector
#include<list>          // std::list
#include<algorithm>     // std::sort
#include<random>        // std::random_device, std::mt19937, and
some other random functions
#include<string>        // std::string
#include<fstream>       // std::fstream

#define DATA_FILE "../inputs/"

//The structure of the rectangle
typedef struct rectangle{
    double width;
    double height;
    double x1,y1;

    //Overload the assignment operator
    void operator=(const rectangle& other)
    {
        width = other.width;
        height = other.height;
        x1 = other.x1;
```

```cpp
        y1 = other.y1;
    };
} rectangle;

struct POINT{
    double x;
    double y;
    double width;

    //Overload the equal operator
    bool operator==(const POINT& other)
    {
        return x==other.x && y==other.y && width==other.width;
    };
};

// cmp function
bool cmpRecWidth(rectangle a, rectangle b);
bool cmpRecWeight(rectangle a, rectangle b);

// BL function
double  BL(std::vector<rectangle>*  recs,  double  width,  bool
isDebug, std::string debugFile);
double BL_change(std::vector<rectangle>* recs, double width, bool
isDebug, std::string debugFile);

// Read and generate functions
std::vector<rectangle> readRecs(std::string filename);
std::vector<rectangle>  random_generate(double  maxHeight,double
maxWeight,int num, int mode);
std::vector<rectangle>  square_generate(double   Height,double
Weight,int hNum, int wNum);
void  printrecs(std::vector<rectangle>  recs,  double  maxHeight,
double maxWeight, std::string filename);

#endif
```

5.3 BL.cpp

```cpp
/****************************************************
 * This file is used to implement the BL algorithm and BL_change
algorithm
 *
 * But my implementation of the BL algorithm here does not fully
restore the idea of the algorithm.
 * I did not place the objects as far as possible in the lower
left position.
 * I could only ensure that the y coordinate of each rectangle is
as small as possible
 * and the x coordinate is aligned with the corresponding leftmost
coordinate point.
 *
 * The BL_change algorithm is just adjust the placement logic of
the rectangle based on the BL algorithm
 ***************************************************/
#include"headers/header.h"
#include<queue>
#define Infinite 1000000000
using namespace std;

// sort the recs by width, from large to small
bool cmpRecWidth(rectangle a, rectangle b)
{
    return a.width > b.width;
}
// sort the recs by weight, from large to small
// The weight is defined as the height divided by the width
// In this way, we can arrange the thin rectangles as early
as possible
bool cmpRecWeight(rectangle a, rectangle b)
{
    return a.height/a.width > b.height/b.width;
}

//"*recs":  The  address  passed  in  is  to  store  coordinate
information
```

```cpp
double BL(vector<rectangle> *recs, double width, bool isDebug,
string debugFile)
{
    ofstream debug;
    debug.open(debugFile,ios::out);

    //initialize
    //Using list to store the upBound, because we need to insert
and delete elements frequently
    list<POINT> upBound;
    //upBound only record the left up point of the rectangle
    //The storage order of upBound is from left to right
    upBound.push_back({width,0,0});//the bound of the rightmost
    upBound.push_back({0,0,width});//the initial point
    int maxHeight=0;

    //sort the recs by width, from large to small
    sort((*recs).begin(),(*recs).end(),cmpRecWidth);

    //Place rectangles in order
    int cnt = 0;
    while(cnt < (*recs).size())
    {
        //initialize
        double height=Infinite;
        auto rightPoint = upBound.begin();
        int pointCase=0;

        //Step 1: Find the shortest rec that meets the requirements.
        //        If they are equal, the leftmost one takes priority.
        for(auto p=upBound.begin(); p≠upBound.end(); p++)
        {
            //If placed at point p, bound means the left boundary
            double bound = (*p).x + (*recs)[cnt].width;
            //Case1: the width of the rec is less than the width
of the current point
            if((*recs)[cnt].width ⩽ (*p).width)
            {
                //If the height of the current point is lower
```

```cpp
than the previous one
            if(height > (*p).y)
            {
                //Update the height of the rec
                height = (*p).y;
                rightPoint = p;
                pointCase = 1;
            }
        }
        //Case2: the rec can be placed in the current point
        else if (bound ≤ width)
        {
            //If there is no rec that can block the rec, the
flag is true
            bool flag=true;
            //Check if there is a rec that can block the rec
            for(auto it=p; it≠upBound.begin(); it--)
            {
                if((*it).x < bound)
                {
                    if((*it).y > (*p).y)
                    {
                        flag = false;
                        break;
                    }
                }
            }
            //If there is no rec that can block the rec,
             //and the height of the current point is lower
than the previous one
            if(flag)
            {
                if(height > (*p).y)
                {
                    height = (*p).y;
                    rightPoint = p;
                    pointCase = 2;
                }
            }
```

```cpp
        }

    }

        //Step 2: Place rec according to the situation and
update upBound
        auto p = rightPoint;
        double bound = (*p).x + (*recs)[cnt].width;
        //Update Boundary

        //Case 1:
        if(pointCase==1)
        {
            //Prevent zero-width lines from appearing
            if((*p).width-(*recs)[cnt].width > 0)
                    upBound.insert(p,{(*p).x+(*recs)[cnt].width,
(*p).y, (*p).width-(*recs)[cnt].width});
            upBound.insert(p, {(*p).x, (*p).y+(*recs)[cnt].height,
(*recs)[cnt].width});
            //If the rec is higher, update maxHeight
            if((*p).y + (*recs)[cnt].height > maxHeight)
                maxHeight = (*p).y + (*recs)[cnt].height;

            //update the rec location
            (*recs)[cnt].x1 = (*p).x;
            (*recs)[cnt].y1 = (*p).y;
            cnt++;

            //delete the current point
            upBound.remove(*p);
        }
        //Case2
        else if(pointCase==2)
        {
            //store the point that need to be deleted
            vector<POINT> kill;
            //Check all points on the right side in turn
            for(auto it=p; it≠upBound.begin(); it--)
            {
```

```
                //The goal is to find a rectangle that is not
completely covered
                //or a point that is just not covered.
                if((*it).x < bound)
                {
                    //Not fully covered rec
                    if ((*it).x + (*it).width ≥ bound)
                    {
                        //update the rec location
                        (*recs)[cnt].x1 = (*p).x;
                        (*recs)[cnt].y1 = (*p).y;
                        if((*p).y + (*recs)[cnt].height > maxHeight)
                            maxHeight = (*p).y + (*recs)[cnt].height;

                        //Prevent zero-width lines from appearing
                        if((*it).x+(*it).width-bound > 0)
                            upBound.insert(p,{bound, (*it).y,
(*it).x+(*it).width-bound});
                            upBound.insert(p,{(*p).x, (*p).y+(*recs)
[cnt].height, (*recs)[cnt].width});
                        kill.push_back(*it);
                        break;
                    }
                    else
                        kill.push_back(*it);
                }
                //The point that is just not covered
                else if((*it).x == bound)
                {
                    //update the rec location
                    (*recs)[cnt].x1 = (*p).x;
                    (*recs)[cnt].y1 = (*p).y;
                    if((*p).y + (*recs)[cnt].height > maxHeight)
                        maxHeight = (*p).y + (*recs)[cnt].height;
                        upBound.insert(p,{(*p).x, (*p).y+(*recs)
[cnt].height, (*recs)[cnt].width});
                    it--;
                    break;
                }
```

```cpp
                    else
                        break;
                }
                //delete the point that need to be deleted
                for(auto it=kill.begin(); it≠kill.end(); it++)
                {
                    //If debug mode is turned on, output the information
of the deleted points
                    if(isDebug)
                        debug << "kill.x: " << (*it).x << " kill.y:
" << (*it).y << " width: " << (*it).width << endl;
                    upBound.remove(*it);
                }
                if(isDebug)
                    debug << endl;
                cnt++;
            }
            else cnt++;
            //If debug mode is turned on, output the information of
the upBound step by step
            if(isDebug)
            {
                debug << "cnt:" << cnt << endl;
                for(auto it=upBound.begin(); it≠upBound.end(); it++)
                    debug << "x: " << (*it).x << " y: " << (*it).y <<
" width: " << (*it).width << endl;
                debug << endl;
            }

        }

        //If debug mode is turned on, output the information of the
all the recs
        if(isDebug)
        {
            debug << "maxHeight: " << maxHeight << endl;
            for(int i=0; i<(*recs).size(); i++)
                debug << "x1: " << (*recs)[i].x1 << " y1: " <<
(*recs)[i].y1 << " width: " << (*recs)[i].width << " height: " <<
```

```cpp
        (*recs)[i].height << endl;
    }

    return maxHeight;
}

//"*recs": The address passed in is to store coordinate
information
double BL_change(vector<rectangle> *recs, double width, bool
isDebug, string debugFile)
{
    ofstream debug;
    debug.open(debugFile,ios::out);

    //initialize
    //Using list to store the upBound, because we need to insert
and delete elements frequently
    list<POINT> upBound;
    //upBound only record the left up point of the rectangle
    //The storage order of upBound is from left to right
    upBound.push_back({width,0,0});//the bound of the rightmost
    upBound.push_back({0,0,width});//the initial point
    int maxHeight=0;

    //Classify the rectangles,
     //putting all those greater than half the width into one
category,
    //and the rest into another category
    vector<rectangle> wideRecs, narrowRecs;
    for(int i=0; i<(*recs).size(); i++)
    {
        if((*recs)[i].width ≥ width/2)
            wideRecs.push_back((*recs)[i]);
        else
            narrowRecs.push_back((*recs)[i]);
    }
    //sort by width(decreasing order)
    sort(wideRecs.begin(),wideRecs.end(),cmpRecWidth);
    //sort by weight(increasing order)
```

```cpp
    sort(narrowRecs.begin(),narrowRecs.end(),cmpRecWeight);

    //Place wideRecs in order
    int cnt = 0;
    while(cnt < wideRecs.size())
    {
        //Due to their properties, it is only possible to place
it on the far left.
        //So put them is easy and fast
        auto p1 = upBound.end();
        p1--;
        if((*p1).width-wideRecs[cnt].width > 0)
                upBound.insert(p1,{(*p1).x+wideRecs[cnt].width,
(*p1).y, (*p1).width-wideRecs[cnt].width});
        upBound.insert(p1,{(*p1).x,(*p1).y+wideRecs[cnt].height,
wideRecs[cnt].width});
        upBound.remove(*p1);

        wideRecs[cnt].x1 = (*p1).x;
        wideRecs[cnt].y1 = (*p1).y;
        cnt++;

        //If debug mode is turned on,
        //output the information of the upBound step by step
        if(isDebug)
        {
            debug << "cnt:" << cnt << endl;
            for(auto it=upBound.begin(); it≠upBound.end(); it++)
            {
                debug << "x: " << (*it).x << " y: " << (*it).y <<
" width: " << (*it).width << endl;
            }
            debug << endl;
        }
    }

    //Place narrowRecs in order
    cnt = 0;
    while(cnt < narrowRecs.size())
```

38

```cpp
    {
        //initialize
        double height=Infinite;
        auto rightPoint = upBound.begin();
        int pointCase=0;

        //same as BL, first find the proper place
        for(auto p=upBound.begin(); p≠upBound.end(); p++)
        {
            double bound = (*p).x + narrowRecs[cnt].width;
            //case1:
            if(narrowRecs[cnt].width ≤ (*p).width)
            {
                //If the height of the current point is lower
than the previous one
                if(height > (*p).y)
                {
                    //Update the height of the rec
                    height = (*p).y;
                    rightPoint = p;
                    pointCase = 1;
                }
            }
            //case2:
            else if (bound ≤ width)
            {
                //If there is no rec that can block the rec, the
flag is true
                bool flag=true;
                //Check if there is a rec that can block the rec
                for(auto it=p; it≠upBound.begin(); it--)
                {
                    if((*it).x < bound)
                    {
                        if((*it).y > (*p).y)
                        {
                            flag = false;
                            break;
                        }
                    }
```

```cpp
                }
            }
            //If there is no rec that can block the rec,
             //and the height of the current point is lower
than the previous one
            if(flag)
            {
                if(height > (*p).y)
                {
                    height = (*p).y;
                    rightPoint = p;
                    pointCase = 2;
                }
            }
        }

    }

    //simular to BL algorithm,
     //but this time we need to consider Case 3: we need to
add a new line
    auto p = rightPoint;
    double bound = (*p).x + narrowRecs[cnt].width;
    //update the upBound
    //case 1
    if(pointCase==1)
    {
        //Prevent zero-width lines from appearing
        if((*p).width-narrowRecs[cnt].width > 0)
            upBound.insert(p,{(*p).x+narrowRecs[cnt].width,
(*p).y, (*p).width-narrowRecs[cnt].width});
                            upBound.insert(p,   {(*p).x,
(*p).y+narrowRecs[cnt].height, narrowRecs[cnt].width});
        //If the rec is higher, update maxHeight
        if((*p).y + narrowRecs[cnt].height > maxHeight)
            maxHeight = (*p).y + narrowRecs[cnt].height;

        //update the rec location
        narrowRecs[cnt].x1 = (*p).x;
```

```cpp
                narrowRecs[cnt].y1 = (*p).y;
                cnt++;

                upBound.remove(*p);
            }
            //case 2
            else if(pointCase==2)
            {
                vector<POINT> kill;
                //Check all points on the right side in turn
                for(auto it=p; it≠upBound.begin(); it--)
                {
                    if((*it).x < bound)
                    {
                        //Not fully covered rec
                        if ((*it).x + (*it).width ≥ bound)
                        {
                            //update the rec location
                            narrowRecs[cnt].x1 = (*p).x;
                            narrowRecs[cnt].y1 = (*p).y;
                            //If the rec is higher, update maxHeight
                                if((*p).y + narrowRecs[cnt].height
> maxHeight)

                                maxHeight = (*p).y + narrowRecs[cnt].height;

                            if((*it).x+(*it).width-bound > 0)
                                    upBound.insert(p,{bound, (*it).y,
(*it).x+(*it).width-bound});
                                        upBound.insert(p,{(*p).x,
(*p).y+narrowRecs[cnt].height, narrowRecs[cnt].width});
                            kill.push_back(*it);
                            break;
                        }
                        else
                            kill.push_back(*it);
                    }
                    //The point that is just not covered
                    else if((*it).x == bound)
                    {
```

```cpp
                    //update the rec location
                    narrowRecs[cnt].x1 = (*p).x;
                    narrowRecs[cnt].y1 = (*p).y;
                    //If the rec is higher, update maxHeight
                    if((*p).y + narrowRecs[cnt].height > maxHeight)
                        maxHeight = (*p).y + narrowRecs[cnt].height;
                                upBound.insert(p,{(*p).x,
(*p).y+narrowRecs[cnt].height, narrowRecs[cnt].width});
                    it--;
                    break;
                }
                else
                    break;
            }
            //delete the point that need to be deleted
            for(auto it=kill.begin(); it≠kill.end(); it++)
            {
                //If debug mode is turned on, output the information
of the deleted points
                if(isDebug)
                    debug << "kill.x: " << (*it).x << " kill.y:
" << (*it).y << " width: " << (*it).width << endl;
                upBound.remove(*it);
            }
            if(isDebug)
                debug << endl;
            cnt++;
        }
        //Case 3: add a new line
        //(That is, at the current highest height, place the
rectangle on the far left)
        else
        {
            vector<POINT> kill;
            //Record all points that need to be deleted
            for(auto p=upBound.begin(); p≠upBound.end(); p++)
            {
                if((*p).x < narrowRecs[cnt].width)
                {
```

```cpp
                        kill.push_back(*p);
                        if((*p).x+(*p).width-narrowRecs[cnt].width
> 0)
                            upBound.insert(p,{narrowRecs[cnt].width,
(*p).y, (*p).x+(*p).width-narrowRecs[cnt].width});
                    }
                }
            //delete the point that need to be deleted
            debug << "**********************" << endl;//Used to
mark case3
            for(auto it=kill.begin(); it≠kill.end(); it++)
            {
                if(isDebug)
                    debug << "kill.x: " << (*it).x << " kill.y:
" << (*it).y << " width: " << (*it).width << endl;
                upBound.remove(*it);
            }
            if(isDebug)
                debug << endl;
            upBound.push_back({0,maxHeight+narrowRecs[cnt].height,narrowR
            maxHeight += narrowRecs[cnt].height;
            cnt++;
        }
        //If debug mode is turned on, output the information of
the upBound step by step
        if(isDebug)
        {
            debug << "cnt:" << cnt << endl;
            for(auto it=upBound.begin(); it≠upBound.end(); it++)
                debug << "x: " << (*it).x << " y: " << (*it).y <<
" width: " << (*it).width << endl;
            debug << endl;
        }

    }

    //Merge the wideRecs and narrowRecs
    for(int i=0; i<wideRecs.size(); i++)
        (*recs)[i] = wideRecs[i];
```

43

```cpp
    for(int i=0; i<narrowRecs.size(); i++)
        (*recs)[i+wideRecs.size()] = narrowRecs[i];


    //If debug mode is turned on, output the information of the
all the recs
    if(isDebug)
    {
        debug << "maxHeight: " << maxHeight << endl;
        for(int i=0; i<(*recs).size(); i++)
        {
            debug << "x1: " << (*recs)[i].x1 << " y1: " <<
(*recs)[i].y1 << " width: " << (*recs)[i].width << " height: " <<
narrowRecs[i].height << endl;
        }
    }


    return maxHeight;
}
```

5.4 BLMain.cpp

```cpp
#include"headers/header.h"
#include<ctime>
using namespace std;

int main()
{
test:
    int i;
    double width;
    string filename;
    //The file path of the debug information
    const string debugFile = "../outputs/BL/debug/debugfile";
    //The file path of the output rectangles information
    const string recFile = "../outputs/BL/recs/rectangles";

    //open the file
    fstream infile, outfile[2];
    for(i=0; i<1; i++)
```

```cpp
        outfile[i].open(recFile+to_string(i)+".txt", ios::out);

    //input the data test file name
    cout << "\033[34mPlease input the name of the file you want
to test:" << endl;
    cout << "PS: the file should be in the \"data\" folder, and
you need to give the file name including suffix\033[0m" << endl;
    cin >> filename;
    infile.open(DATA_FILE + filename, ios::in);

    infile >> width;

    //Read the data from the file
    //recs[2] is used to store the initial data,
    //which is used to restore the data after each test
    vector<rectangle> recs[3];
    recs[0] = readRecs(filename);
    recs[1] = recs[0];
    recs[2] = recs[0];

        cout << "\033[34mOpen test mode (generate no debug
information)?\033[0m(Y/N)" << endl;
    char c; cin >> c;

    //The test mode:
    if(c == 'Y' || c == 'y')
    {
        //The number of loop executions
            cout << "\033[34mNumber of loop executions:\033[0m"
<< endl;
        int k; cin >> k;

        double solution,duration=0;
        clock_t start,finish;

        //The BL algorithm and the BL_change algorithm are executed
k times respectively
            //The data is restored after each test (Except for the
last test)
```

```cpp
        //Overloaded the rec assignment operation to ensure
deep restore
        for(i=0;i<k;i++)
        {
            //test the BL algorithm
            start = clock();
            solution = BL(recs, width, false, debugFile+"_BL.txt");
            finish = clock();
            duration+=(double)(finish-start)/CLOCKS_PER_SEC;

            //restore the data
            if(i≠k-1)
                recs[0] = recs[2];
        }
        cout << "\033[31mmaxHeight(BL):" << solution << endl;
        printf("Time:%lfs\n\n", duration);

        duration = 0;
        for(i=0; i<k; i++)
        {
            //test the BL_change algorithm
            start = clock();
                solution = BL_change(recs+1, width, false,
debugFile+"_BL_Change.txt");
            finish = clock();
            duration+=(double)(finish-start)/CLOCKS_PER_SEC;

            //restore the data
            if(k≠1 && i≠k-1)
                recs[1] = recs[2];
        }
        cout << "maxHeight(BL_change):" << solution << endl;
        printf("Time:%lfs\033[0m\n\n", duration);
    }
    //debug mode
    else
    {
        cout << "\033[31mmaxHeight(BL):\033[0m" << BL(recs, width,
true, debugFile+"_BL.txt") << endl;
```

46

```cpp
                cout << "\033[31mmaxHeight(BL_change):\033[0m" <<
BL_change(recs+1, width, true, debugFile+"_BL_Change.txt") <<
endl;
    }


    cout << "**********************************" << endl;


    //write the data to the file to draw the figure
    for(i=0; i<2; i++)
    {
        outfile[i] << width << endl;
        for(int j=0; j<recs[i].size(); j++)
            outfile[i] << recs[i][j].x1 <<" "<< recs[i][j].y1 <<"
"<< recs[i][j].width <<" "<< recs[i][j].height << endl;
    }


    /* cout << "generate the fig now?(Y/N)" << endl;
    cout << "\033[34mPS: The program here calls python3 through
the system function. You need to make sure the matplotlib library
is installed.\033[0m" << endl;
    cin >> c;
    if(c=='Y' || c=='y')
    {
        system("python3 scripts/draw.py");
        cout << "Successfully generated the figure! You can see
them in \"fig\" now." << endl;
    } */


    cout << "\033[34mrun again?\033[0m(Y/N)" << endl; cin >> c;
    if(c=='Y' || c=='y') goto test;


    return 0;
}
```

5.5 ttpHeader.h

```cpp
#define ITEMNUM 1000000                                 // Maximum
number of items
#define ITERATIONS 1                                    // Iteration time
```

```cpp
#define INPUTDIR "../inputs/FFDH/inputs/"        // Directory of
the input file
#define OUTPUTDIR "../outputs/FFDH/outputs/"      // Directory of
the output file
#define DRAWINPUTDIR "outputs/FFDH/rects/"       // Input file
for draw.py

// structure of a single item
typedef struct item {
    double width;
    double height;
     double x;              // Position of the upper left point of
the item
    double y;
} Item;

// First Fit by Decreasing Height, a basic 2-approximation
algorithm
// W: Fixed width of the resulting texture
// n: The number of items
// rect: Items
// isDebug: Flag of debug mode
// outFile: Flag of file output mode
int FFDH(double W, int n, Item rect[], int isDebug, int outFile);
```

5.6 FFDH.cpp

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "headers/ttpHeader.h"

double curWidth[ITEMNUM];                    // Array recording the
total item width in each level
int pos[ITEMNUM];                            // Array recording the
position(which level) of every item
double levelHeight[ITEMNUM];                  // Height of each
level(depending on the first item in the level)
int level;                                   // Current level of
```

```cpp
the resulting texture
extern std::string outFileName;                    // Name of the
output file

int cmp(const void * a, const void * b);                      //
Comparator for decrease-order quicksort function
void  printDebugInfo(double  W,  int  n,  Item  *  rect,  int
outFile);    // Print the debug infomation when using "--debug"
command argument

// First Fit by Decreasing Height, a basic 2-approximation
algorithm
// W: Fixed width of the resulting texture
// n: The number of items
// rect: Items
// isDebug: Flag of debug mode
// outFile: Flag of file output mode
int FFDH(double W, int n, Item rect[], int isDebug, int outFile) {
    double curHeight = 0;                    // Current height of
the resulting texture
    int i, j;

    // Initially, sort all items by their heights in decreasing
order
    qsort(rect, n, sizeof(rect[0]), cmp);

    // initialize elements in curWidth to zero
    level = 0;
    for (i = 0; i < n; i++)
        curWidth[i] = 0;

    // Handle all items
    for (i = 0; i < n; i++) {
        // Find the first fit existing level
        for (j = 0; j < level; j++) {
            // Find it!
            if (curWidth[j] + rect[i].width ≤ W) {
                rect[i].x = curWidth[j];            // Update
the position of the item
```

```
                    rect[i].y = levelHeight[j];
                    curWidth[j] += rect[i].width;        // Update
the width of current level
                    pos[i] = j;                          // Record
the position of the item
                break;
            }
        }
        if (j < level)                               // If found,
continue to process next loop
            continue;

        // If not found
        ++level;                                     // Create a
new level
        curWidth[level - 1] = rect[i].width;     // Update the
width of current level
        levelHeight[level - 1] = curHeight;      // Update the
height of the level
        curHeight += rect[i].height;             // Update the
current height
        rect[i].x = 0;                           // Update the
position of the item
        rect[i].y = levelHeight[level - 1];
        pos[i] = level - 1;                      // Record the
level of the item
    }

    // Print the debug infomation when using "--debug" command
argument
    if (isDebug)
        printDebugInfo(W, n, rect, outFile);

    // Return the current height of the resulting texture as the
minimal height
    return curHeight;
}


// Comparator for decrease-order quicksort function
```

```c
int cmp(const void * a, const void * b) {
    const Item dataA = *(const Item*)a;
    const Item dataB = *(const Item*)b;

    // Sort items by their height in decreasing order
    if (dataB.height < dataA.height) {
        return -1;
    } else if (dataB.height > dataA.height) {
        return 1;
    } else {      // equalf
        return 0;
    }
}


// Print the debug infomation when using "--debug" command
argument
// W: Fixed width of the resulting texture
// n: The number of items
// rect: Items
// outFile: Flag of file output mode
void printDebugInfo(double W, int n, Item * rect, int outFile) {
    int i;

    if (!outFile) {      // Default output mode(print the info in
the terminal)
        printf("Debug Info:\n");

        // 1. Print the height-decreasingly sorted item data
        printf("Height-decreasingly sorted item data:\n");
        for (i = 0; i < n; i++) {
                printf("%d: %.2f, %.2f\n", i, rect[i].width,
rect[i].height);
            // ++rect;
        }

        // 2. Print the occupied-by-items width for each level
        printf("\nTotal level: %d\nWidth:\n", level);
        for (i = 0; i < level; i++) {
            printf("%d level: %.2f\n", i, curWidth[i]);
```

```cpp
        }

        // 3. Print the positions of items
        printf("\nPosition:\n");
        for (i = 0; i < n; i++) {
            printf("Item %d: level %d, x: %.2f, y: %.2f\n", i +
1, pos[i], rect[i].x, rect[i].y);
        }

        // Deviding line
        printf("===================================\n");
    } else {    // File output mode
        std::string outDirName;
        outFileName = outFileName == "" ? "output" : outFileName;
        outDirName = OUTPUTDIR + outFileName;
        FILE * fp = fopen(outDirName.c_str(), "w");

        fprintf(fp, "Debug Info:\n");

        // 1. Print the height-decreasingly sorted item data
        fprintf(fp, "Height-decreasingly sorted item data:\n");
        for (i = 0; i < n; i++) {
            fprintf(fp, "%d: %.2f, %.2f\n", i, rect[i].width,
rect[i].height);
            // ++rect;
        }

        // 2. Print the occupied-by-items width for each level
        fprintf(fp, "\nTotal level: %d\nWidth:\n", level);
        for (i = 0; i < level; i++) {
            fprintf(fp, "%d level: %.2f\n", i, curWidth[i]);
        }

        // 3. Print the positions of items
        fprintf(fp, "\nPosition:\n");
        for (i = 0; i < n; i++) {
            fprintf(fp, "Item %d: level %d, x: %.2f, y: %.2f\n",
i + 1, pos[i], rect[i].x, rect[i].y);
        }
```

```cpp
        // Deviding line
        fprintf(fp, "==================================\n");

        fclose(fp);

        // 4.(only for file output) Get the input file for draw.py
                std::string   drawFileName   =   "rectangle"   +
outFileName.substr(6) + ".txt";
        std::string drawDirName = DRAWINPUTDIR + drawFileName;
        fp = fopen(drawDirName.c_str(), "w");

        fprintf(fp, "%.2f\n", W);
        for (i = 0; i < n; i++) {
                fprintf(fp, "%.2f %.2f %.2f %.2f\n", rect[i].x,
rect[i].y, rect[i].width, rect[i].height);
        }

        fclose(fp);

                printf("The  output  file  is  saved  as  %s\n",
outDirName.c_str());      // Hint
    }
}
```

5.7 ttpMain.cpp

```cpp
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <time.h>
#include "headers/ttpHeader.h"


double Width;                    // Fixed width of the resulting texture
int n;                           // The number of items
Item rect[ITEMNUM];              // Items
double totalHeight;              // The sum of all heights of items
```

```cpp
double Area;                    // Area of all items
int isDebug;                    // Flag of debug mode
int isTiming;                   // Flag of timing mode
int outChoice;                  // Flag of file output mode
clock_t start, stop;            // Record of start and stop time of
the approximation algorithm
std::string outFileName = "";     // Name of the output file

// Input handler
void getInput(int argc, char * argv[]);
// Print the timing infomation
void printTime(clock_t start, clock_t end, int outFile);

int main(int argc, char * argv[]) {
    int i;
    double miniHeight;    // Result

    getInput(argc, argv);                                    // Input

    // Execution of approximation algorithm
    if (isTiming) {
        start = clock();                                  //
Start timing
        for (i = 0; i < ITERATIONS; i++)                  //
Multiple execution of algorithms
            miniHeight = FFDH(Width, n, rect, 0, outChoice);    //
Execute!
        stop = clock();                                   //
Stop timing
        printTime(start, stop, outChoice);                //
Print the timing infomation
    } else {
        miniHeight = FFDH(Width, n, rect, isDebug, outChoice);   //
Execute!
        // Output
        if (!outChoice) {
            printf("The total height: %.2f\n", totalHeight);
                printf("The ideal height(Area / Width): %.2f\n",
Area / Width);
```

```cpp
                    printf("The \"minimum\" height: %.2f\n",
miniHeight);    // Output the result in the terminal
        } else {        // Output the result in the file
            std::string outDirName = OUTPUTDIR + outFileName;

            FILE * fp = fopen(outDirName.c_str(), "r");
            if (fgetc(fp) == 'T') {        // If the file contains
the info of last execution
                fclose(fp);
                FILE * fp = fopen(outDirName.c_str(), "w");        //
Clean the original content in the file
            } else {                        // Otherwise, the file
is just empty or contains the debug info
                fclose(fp);
                FILE * fp = fopen(outDirName.c_str(), "a");        //
Append the output to the file
            }
                    fprintf(fp, "The total height: %.2f\n",
totalHeight);
            fprintf(fp, "The ideal height(Area / Width): %.2f\n",
Area / Width);
                    fprintf(fp, "The \"minimum\" height: %.2f\n",
miniHeight);    // Output the result in the file
            fclose(fp);
        }
    }


    return 0;
}

// Input handler
void getInput(int argc, char * argv[]) {
    int i;
    int choice = 0;    // Choice whether receiving terminal input
or file input

    // If using command arguments
    if (argc > 1) {
        for (i = 1; i < argc; i++) {
```

```cpp
                // Handle debug mode
                if (!strcmp(argv[i], "-d") || !strcmp(argv[i],
"--debug"))
                    isDebug = 1;
                // Handle file input mode
            else if (!strcmp(argv[i], "-if") || !strcmp(argv[i],
"--infile"))
                    choice = 1;
            // Handle file output mode
            else if (!strcmp(argv[i], "-of") || !strcmp(argv[i],
"--outfile"))
                    outChoice = 1;
            // Handle timing mode
             else if (!strcmp(argv[i], "-t") || !strcmp(argv[i],
"--timing"))
                    isTiming = 1;
        }
    }

    if (!choice) {      // Terminal input
        scanf("%lf", &Width);
        scanf("%d", &n);
        for (i = 0; i < n; i++) {
            scanf("%lf%lf", &rect[i].width, &rect[i].height);
            totalHeight += rect[i].height;
            Area += rect[i].height * rect[i].width;
        }
    } else {            // File input
        std::string inFileName;           // Name of the input file
        std::string inDirName;            // The whole directory
of the input file

        printf("Please input the input file name:\n");
        std::cin >> inFileName;
        inDirName = INPUTDIR + inFileName;
        outFileName = "output" + inFileName.substr(5);


        FILE * fp = fopen(inDirName.c_str(), "r");       // Read
```

```c
the input file
        if (fp == NULL) {       // If it can't open the file, exit
the program
            printf("Fail to read the input file. Please ensure
that you use the true directory.");
            exit(1);
        }
        fscanf(fp, "%lf", &Width);      // Similar to terminal input
        fscanf(fp, "%d", &n);
        for (i = 0; i < n; i++) {
            fscanf(fp, "%lf%lf", &rect[i].width, &rect[i].height);
            totalHeight += rect[i].height;
            Area += rect[i].height * rect[i].width;
        }

        fclose(fp);
    }
}

// Print the timing infomation
void printTime(clock_t start, clock_t end, int Outfile) {
    clock_t tick;       // Ticks
    double duration;    // Duration(unit: seconds)
    int iterations;

    iterations = ITERATIONS;    // Set iteration time, for obvious
timing result
    tick = end - start;         // Calculate tick numbers
    duration = ((double)(tick)) / CLOCKS_PER_SEC;       // Calculate
the total duration of multiple execution of the algorithm

    // Print the timing info
    if (!Outfile) {     // Default output mode(print the info in
the terminal)
        printf("\nTiming Result:\n");
        printf("Iterations: %d\n", iterations);
        printf("Ticks: %lu\n", (long)tick);
        printf("Duration: %.6fs\n", duration);
    } else {            // File output mode
```

```
        FILE * fp = fopen(OUTPUTDIR, "w");
        if (fp == NULL) {     // If it can't open the file, exit
the program
            printf("Fail to open the output file. Please ensure
that you use the true directory.");
            exit(1);
        }
        fprintf(fp, "\nTiming Result:\n");
        fprintf(fp, "Iterations: %d\n", iterations);
        fprintf(fp, "Ticks: %lu\n", (long)tick);
        fprintf(fp, "Duration: %.6fs\n", duration);

        fclose(fp);
    }
}
```

5.8 generate.cpp

```
/*****************************
 * The four programs here are :
 * 1. scale data generation program
 * 2. optimal solution determination data generation program
 * 3. data recording program
 * 4. data reading program
 ****************************/
#include"headers/header.h"
using namespace std;

//The four parameters are
// maxHeight: the maximum height of the rectangle
// maxWeight: the maximum width of the rectangle
// num: the number of rectangles
// mode: the mode of the distribution
vector<rectangle>   random_generate(double    maxHeight,double
maxWeight,int num, int mode)
{
    vector<rectangle> recs;
    //make sure the random number is different every time
    random_device rd;
```

```cpp
    //"mt19937" is a random number generator: Mersenne Twister
19937 generator
    mt19937 gen(rd());

    //mode 0: uniform distribution(int)
    if(mode == 0)
    {
        //uniform_int_distribution: generate random integers
uniformly distributed on a range
        uniform_int_distribution<> height(1, maxHeight);
        uniform_int_distribution<> weight(1, maxWeight);
        for(int i=0; i<num; i++)
                          recs.push_back({(double)weight(gen),
(double)height(gen),0,0});
    }
    //mode 1: normal distribution(int)
    else if(mode == 1)
    {
        //normal_distribution: generate random numbers according
to a normal distribution
        //static_cast: convert a value to a specified type
        //maxHeight/2: the mean value of the normal distribution
        //maxHeight/6: the standard deviation of the normal
distribution
        //There is no particular reason why the standard deviation
is set this way...
            normal_distribution<> height(static_cast <double>
(maxHeight/2), static_cast <double> (maxHeight/6));
            normal_distribution<> weight(static_cast <double>
(maxWeight/2), static_cast <double> (maxWeight/6));
        for(int i=0; i<num; i++)
        {
            int h = static_cast <int> (height(gen));
            int w = static_cast <int> (weight(gen));

            //make sure the height and width are in the range
            if(h <= 0) h = 1;
            else if(h > maxHeight) h = maxHeight;
            if(w <= 0) w = 1;
```

```cpp
            else if(w > maxWeight) w = maxWeight;

            recs.push_back({(double)w,(double)h,0,0});
        }
    }
    //mode 2: uniform distribution(double)
    else if(mode == 2)
    {
        // similar to int case
        uniform_real_distribution<> height(1, maxHeight);
        uniform_real_distribution<> weight(1, maxWeight);
        for(int i=0; i<num; i++)
            recs.push_back({weight(gen),height(gen),0,0});
    }
    //mode 3: normal distribution(double)
    else if(mode == 3)
    {
        // similar to int case
            normal_distribution<> height(static_cast <double>
(maxHeight/2), static_cast <double> (maxHeight/6));
            normal_distribution<> weight(static_cast <double>
(maxWeight/2), static_cast <double> (maxWeight/6));
        for(int i=0; i<num; i++)
        {
            double h,w;
            h = height(gen);
            w = weight(gen);

            // there is no need to limit the up range of the height
            if(h <= 0) h = 0.1;
            if(w <= 0) w = 0.1;
            else if(w>maxWeight) w = maxWeight;

            recs.push_back({w,h,0,0});
        }
    }


    return recs;
}
```

```cpp
//The generation algorithm here is to generate certain random
points on the edge of the large rectangle, and then divide the
entire rectangle into n*m blocks
//Height: the height of the large rectangle
//Weight: the width of the large rectangle
//hNum: the number of rows
//wNum: the number of columns
vector<rectangle>        square_generate(double        Height,double
Weight,int hNum, int wNum)
{
    vector<rectangle> recs;
    //make sure the random number is different every time
    random_device rd;
     //"mt19937" is a random number generator: Mersenne Twister
19937 generator
    mt19937 gen(rd());

    //Use uniform distribution
    uniform_int_distribution<> height(1, Height-1);
    uniform_int_distribution<> weight(1, Weight-1);

    //hp: point on the height edge
    //wp: point on the weight edge
    vector<int> hp;
    vector<int> wp;
    //push in the start and end point
    hp.push_back(0);
    wp.push_back(0);
    hp.push_back(Height);
    wp.push_back(Weight);
    while(hp.size()<hNum+1)
    {
        int h = height(gen);
        //make sure the point is different
        if(find(hp.begin(), hp.end(), h) == hp.end())
            hp.push_back(h);
    }
    while(wp.size()<wNum+1)
```

```cpp
    {
        int w = weight(gen);
        //make sure the point is different
        if(find(wp.begin(), wp.end(), w) == wp.end())
            wp.push_back(w);
    }
    //sort the points
    sort(hp.begin(), hp.end());
    sort(wp.begin(), wp.end());

    //generate the blocks
    for(int i=1; i<=hNum; i++)
        for(int j=1; j<=wNum; j++)
                recs.push_back({(double)(wp[j]-wp[j-1]),(double)
(hp[i]-hp[i-1]),0,0});

    return recs;
}

//write the data to the file in the format of the test example
    /*********************
     * @ test example:
     * maxWeight
     * n
     * weight_1 height_1
     * ...
     * weight_n height_n
    *********************/
void printrecs(vector<rectangle> recs, double maxHeight, double
maxWeight, string filename)
{
    fstream outfile;
    outfile.open(DATA_FILE+filename, ios::out);

    outfile << maxWeight << " " << recs.size() << std::endl;
    for(int i=0; i<recs.size(); i++)
            outfile << recs[i].width << " " << recs[i].height
<< std::endl;
```

```cpp
        outfile.close();
}


//read the data from the particular file into a vector<rectangle>
vector<rectangle> readRecs(std::string filename)
{
    fstream infile;
    infile.open(DATA_FILE+filename, ios::in);

    vector<rectangle> recs;
    double maxWeight;
    int n;

    //read the data
    infile >> maxWeight >> n;
    for(int i=0; i<n; i++)
    {
        double w,h;
        infile >> w >> h;
        recs.push_back({w,h,0,0});
    }
    //close the file
    infile.close();
    return recs;
}
```

5.9 generate_main.cpp

```cpp
#include"headers/header.h"
using namespace std;

const string generateFile="../inputs/generate/generate_in.txt";

//For the convenience of input and implementation, only file input
is supported
/***************
 * @ generate_in file format:
 * generateMode(decide the mode of the generation)
 * n(number of test cases you want to generate)
```

```
 *
 * @ generateMode 1
 * maxHeight_1 maxWeight_1 num_1 mode_1 filename_1
 * ...
 * maxHeight_n maxWeight_n num_n mode_n filename_n
 *
 * @ generateMode 2
 * Height_1 Weight_1 hNum_1 wNum_1 filename_1
 * ...
 * Height_n Weight_n hNum_n wNum_n filename_n
 ****************/
int main()
{
    fstream infile;
    infile.open(generateFile, ios::in);

    int generateMode;
    infile >> generateMode;

    //generateMode1: random generation
    if(generateMode==1)
    {
        //mode 0: uniform int distribution
        //mode 1: normal distribution(int)
        //mode 2: uniform real distribution
        //mode 3: normal distribution(double)
        int i;
        double maxHeight,maxWeight;
        int num,mode,n;
        string filename;

        infile >> n;
        //generate n test cases
        for(i=0; i<n; i++)
        {
            infile >> maxHeight >> maxWeight >> num >> mode;
            infile >> filename;

            vector<rectangle> recs;
```

```cpp
            //generate the rectangles
            recs = random_generate(maxHeight, maxWeight, num, mode);
            //write the data to the file
            printrecs(recs, maxHeight, maxWeight, filename);
        }
    }
    //generateMode2: square generation
    else if(generateMode==2)
    {
        int i;
        double Height,Weight;
        int hNum,wNum,n;
        string filename;

        //generate n test cases
        infile >> n;
        for(i=0; i<n; i++)
        {
            infile >> Height >> Weight >> hNum >> wNum;
            infile >> filename;

            vector<rectangle> recs;
            //generate the rectangles
            recs = square_generate(Height, Weight, hNum, wNum);
            //write the data to the file
            printrecs(recs, Height, Weight, filename);
        }
    }

    return 0;
}
```

References

1. Wikipedia. Strip packing problem, https://en.wikipedia.org/wiki/Strip_packing_problem

2. Brenda S. Baker, E. G. Coffman, JR and Ronald L. Rivest, *Orthogonal Packings in Two Dimensions*, https://epubs.siam.org/doi/abs/10.1137/0209064

3. E. G. Coffman, JR., M. R. Garey, D. S. Johnson, and R. E. Tarjan, *Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms*, https://epubs.siam.org/doi/abs/10.1137/0209062

Declaration

We hereby declare that all the work done in this project titled "Texture Packing" is of our independent effort as a group.