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.

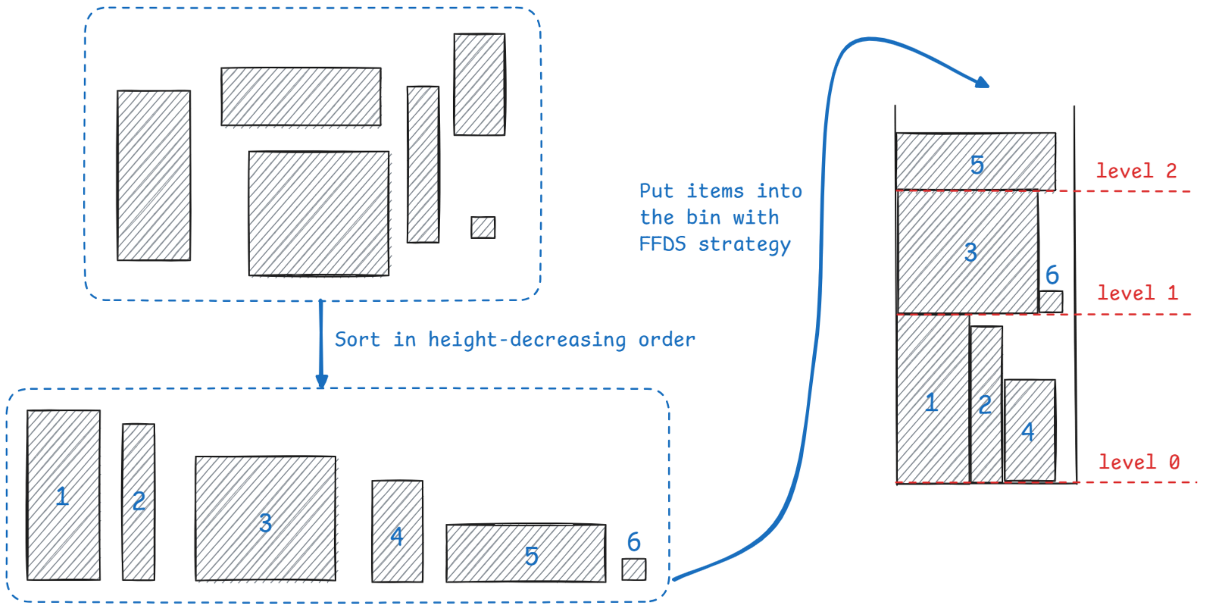
1.2 Backgound of Data Structures and Algorithms

1.2.1 BL Algorithm

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.


Sort in height-decreasing order
Put items into the bin with FFDS strategy
1
2
3
4
5
6
level 2
level 1
level 0
Figure 1: FFDH Approximation Algorithm

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.

Beware that our project doesn't use complex data structures and we only use arrays and structures in C, so it's meaningless to introduce them and we should only focus on the algorithm implementation.

2.1 BL Algorithm

2.2 FFDH Algorithm

Initially, 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**)

```
Begin
    Sort rect[] by item's height in decreasing order
    for item in rect[] do
        for level in existing levels do
            if level's width + item[i]→width ≤ W then
                put the item into this level and update the state
                break
            end
        if no level can fit the item then
            create a new level
            put the item into this level
            update the state
        end
    end
    print debug info if the user using the debug mode
    return the current height of the bin as the "minimum" height
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.

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

3.1.2 Performance Tests

3.2 FFDH Algorithm

3.2.1 Correctness Tests

Test 1

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

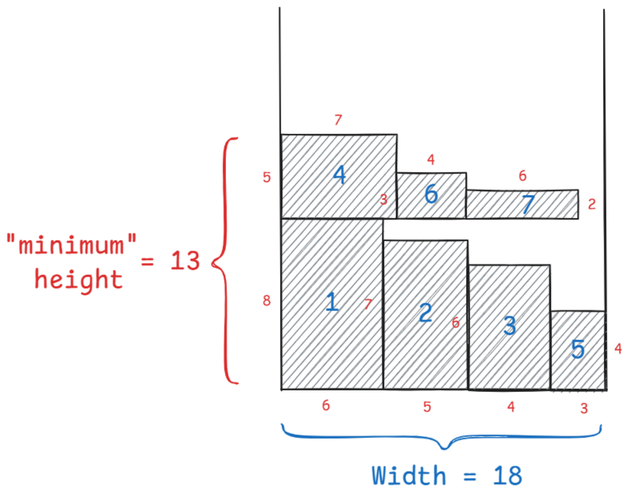


Figure 2: Correctness Test 1 for FFDH Algorithm

- Actual Result:

./code/test/FFDH/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 0: level 0
Item 1: level 0
Item 2: level 0
Item 3: level 1
Item 4: level 0
Item 5: level 1
Item 6: level 1
====================================
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.

Test 2

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

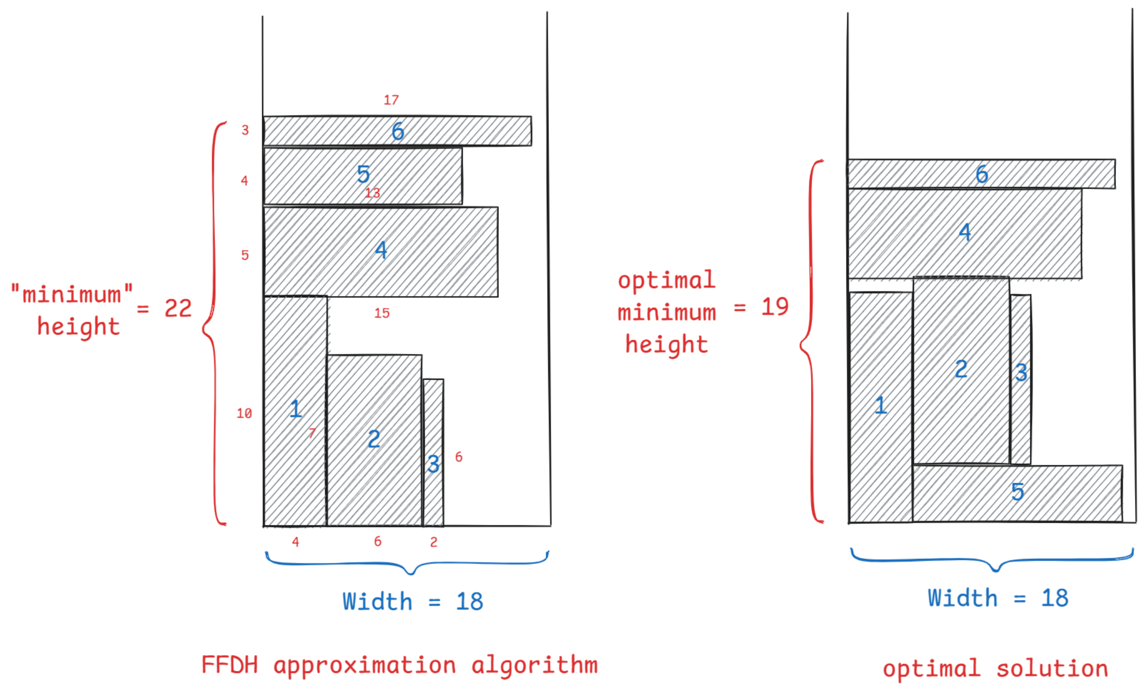


Figure 3: Correctness Test 2 for FFDH Algorithm

- Actual Result:

```
./code/test/FFDH/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 0: level 0
Item 1: level 0
Item 2: level 0
Item 3: level 1
Item 4: level 2
Item 5: level 3
====================================
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.

Test 3

- Purpose: check the correctness in **the large-scale case, with very large waste space**.
- Input: See the input file in directory `./code/test/FFDH/input3`
- Expected Result:

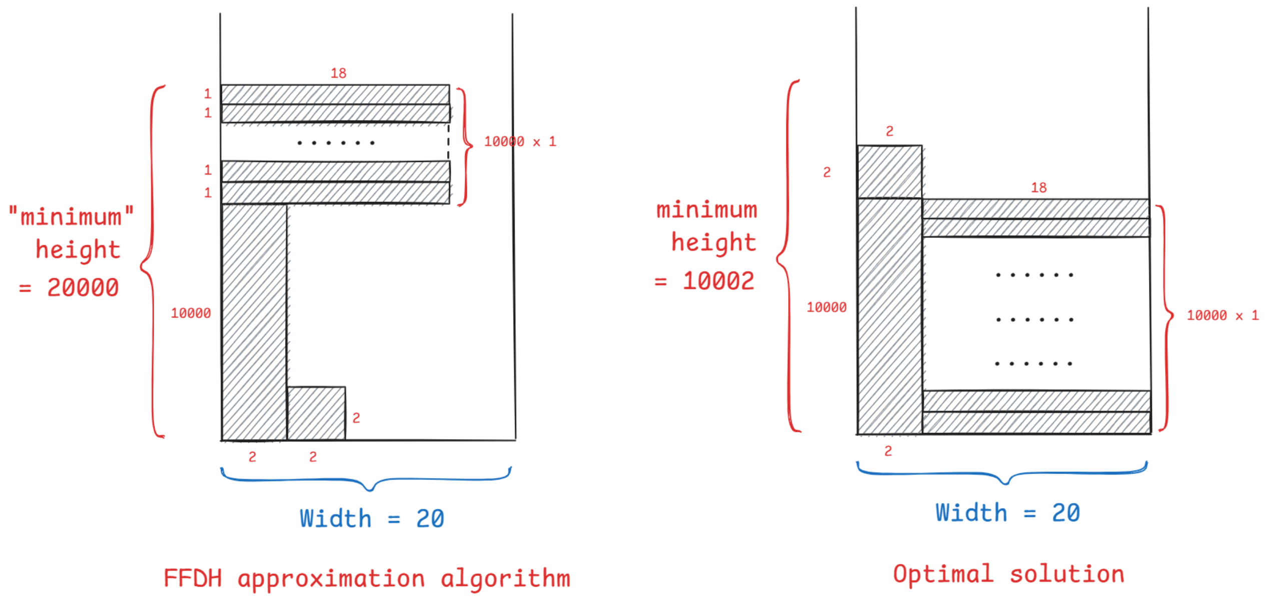


Figure 4: Correctness Test 3 for FFDH Algorithm

- Actual Result:

```
./code/test/FFDH/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
Item 1: level 0
Item 2: level 1
Item 3: level 2
...
Item 10000: level 9999
Item 10001: level 10000
====================================
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 $\frac{FFGH(L)}{OPT(L)} \approx 2$, 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 uniform distribution, 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 1: 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.

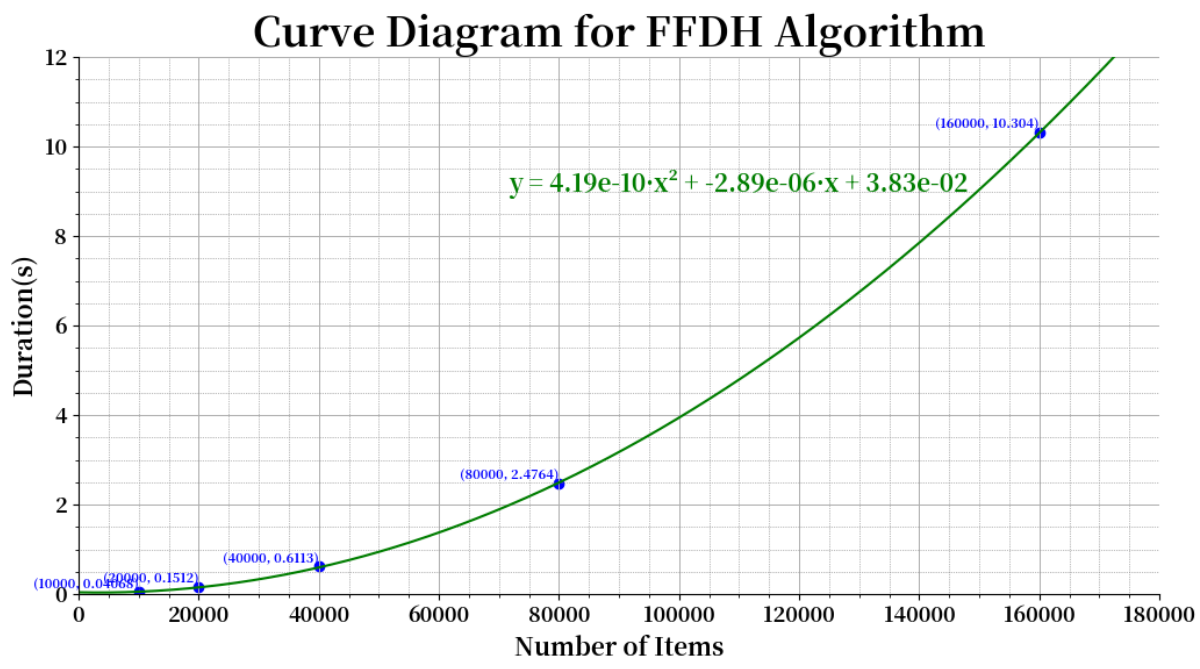


Figure 5: Curve Diagram for FFDH Algorithm in Performance Test

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.

Chapter 4: Analysis and Comments

4.1 Space Complexity

Conclusion:

- FFDH algorithm: $O(N)$, N is the number of items.
- advanced approximation algorithm:

Analysis:

- 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.

- advanced approximation algorithm:

4.2 Time Complexity

Conclusion:

- FFDH algorithm: $O(N^2)$, N is the number of items.
- advanced approximation algorithm:

Analysis:

- 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 $O(N^2)$
- advanced approximation algorithm:

4.3 Further Improvement

Appendix: Source code

5.1 File Structure

```
.
├── code
│   ├── README.md
│   ├── build
│   │   └── Makefile
│   ├── scripts
│   │   ├── generate.cpp
│    │    ├── ttpHeader.h
│    │    ├── ttpMain.cpp
│    │    ├── FFDH.cpp
│    │    ├── curve.py
│    │    └──
│    │
│    └── test
│         ├── FFDH
│         │    ├── input1
│         │    ├── input2
│         │    ├── input3
│         │    ├── output1
│         │    ├── output2
│         │    └── output3
│         └──
│
└── documents
     └── report.pdf
```

5.2 ttpHeader.cpp

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

5.3 ttpMain.cpp

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <time.h>
#include "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
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);
                     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++) {
            // 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
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);
    }
}
```

5.4 FFDH.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "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
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
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]);
        }

        // 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);
        }

        // 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
    }
}
```

References

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

Declaration

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