Advanced Data Structure And Algorithm

Projects 6: Texture Packing

Author:

Date: 2024/11/30

2024-2025 Autumn & Winter Semester

Table of Contents

Chapter 1: Introduction

Problem: Texture Packing is to pack multiple rectangle shaped textures into one large texture. The resulting texture must have a given width and a minimum height.

Apparently, this problem is a 2-dimensional version of Binpacking Problem, And accordingly, we got the inspiration of implementing the according algorithms which are traditionally useful.

Therefore, several algorithms will be presented and discussed in this report. In Chapter 2, we will introduce the traditional algorithms, and a finally revised one. The correctness and performance of those algorithms are tested and measured in Chapter 3, and in Chapter 4, we will give a thorough analysis on all the factors that might affect the approximation ratio of our proposed algorithm

Chapter 2: Algorithm Specification

2.1) Next-Fit Algorithm((NF)

Imagine a big rectangle ready for us to fit items in, whose width equals to the given width, and height can be infinite. We start from the bottom level of the texture and place the first rectangle item in the leftmost position. Then we define the rest of the space on the level as a block, which is okay to put another item in. For every new item, if the block can accommodate it, we put it in and update the width and height of the new block; if not, we put it on the new level and build a new block.

Pseudocode:

```
Main Function:

  Allocate memory for the item1 array
  For each item:
    Read the item's width and height


  Initialize the first block:
    block[1].width = W - item1[1].width
    block[1].height = item1[1].height
    Initialize container height H = item1[1].height

  Initialize index i = 2 and decrement N by 1

  While there are items left:
    If block[1] can fit item[i]:
      Update block[1]'s dimensions after placing item[i]
    Else:
      Create a new block for item[i]
      Update container height H
    Increment i and decrement N
```

```
Output the final height H and elapsed time
```

Time Complexity: Obviously $O(N)$, since there is only one loop with operations taking linear time.

2.2) Next-Fit Decreasing Algorithm(NFD)

To maximize the utilization of space, we sort the items by their heights in non-increasing order, then implement the same way we treat each item in the next-fit algorithm. In this way, there will be less space wasted between each level since the heights of items on each level are most close.

Pseudocode:

```
Main Function:

  Perform heap sort on item1

  Allocate memory for block array and initialize the first block:
  Remove the first item from item1 (pop)
  Reduce N by 1

  While there are still items left:
    If the current block can fit the first item:
      Update block width and height after placing the item
    Else:
      Create a new block for the item
      Update container height H
    Remove the first item from item1 (pop)
    Reduce N by 1



  Output the final height H and elapsed time
```

Time Complexity: In the main function, the dominant operation in terms of time complexity is heap sort operation with a time complexity of $O(N\log(N))$, and the while loop that runs as long as there are remaining items (N $>$ 0). Inside the while loop, there are constant time operations like comparisons and simple arithmetic, along with a call to the pop function in each iteration which has a time complexity of $O(\log(N))$. Since the loop runs N times in the worst case, the time complexity contributed by the while loop is also $O(N\log(N))$.

2.3) Best-Fit Decreasing Algorithm Advanced(BFDA)

Now we make an improvement. The usual best-fit, which put the rest of the spaces into a block array, and choose them by width, is not the best choice, compared to next fit decreasing, there exits a better choice. We can ultilize the remaining space of the last level, all the remaining space, not just the right-most block. Everytime we insert an item in a block, two other blocks pop up from the upper space of the item and the right space of it. We add those two blocks into the block array, and delete the previous block. Then we choose the smallest block to put the next item in.(This time we can use height to determine the fittest block, because of the fact that some of the height of the blocks are determined by a previous block minus a item's height)

Pseudocode:

```
Main Function:

  Allocate memory for item1 array and read the width and height of each item

  Perform heap sort on item1

  Allocate memory for block array and initialize the first block:
    block[1].width = W - item1[1].width
    block[1].height = item1[1].height
    Initialize container height H = item1[1].height
    Remove the first item from item1 (pop)

  Initialize counters:
    amountBlock = 1
    count = 0

  While there are still items left:
    For each block:
      If the current block can accommodate the first item:
        Update the block's width and height
        Create and insert new blocks (splitting the block)
        Update the block count
        Remove the first item from item1
        Break the loop
      If no block can accommodate the item:
        Create a new block and update container height H
        Insert the new block into block array
        Update the block count
        Remove the first item from item1



  Output the final height H and result
```

Time Complexity: Considering nested loop, the flexible time consumed in searching, the time complexity is $O(N) \sim O(N^3)$

Approximation Ratio: The worst case of this algorithm is when there's no posibility to put anything in the small blocks in each level, therefore the approximation ratio is the same with the NFDH and FFDH described on wiki.

Therefore, $\boldsymbol{H \leq 1.7h_{\text{op}} + h_{\max}}$ where h_{op} is the optimal height and $h_{\max}$ is maximal height in these items. Also, note that $\boldsymbol{h_{\max} \leq h_{\text{op}}}$, and the **worst** approximation ratio is 2.7.(quoted from wiki)

Chapter 3: Testing Results

3.1) Correctness

Input	Output
7 18 6 8 5 7 4 6 7 5 3 4 4 3 6 2	13

3.2) Performance

NF: The time complexity is obvious, no need to produce a curve.

NFD:

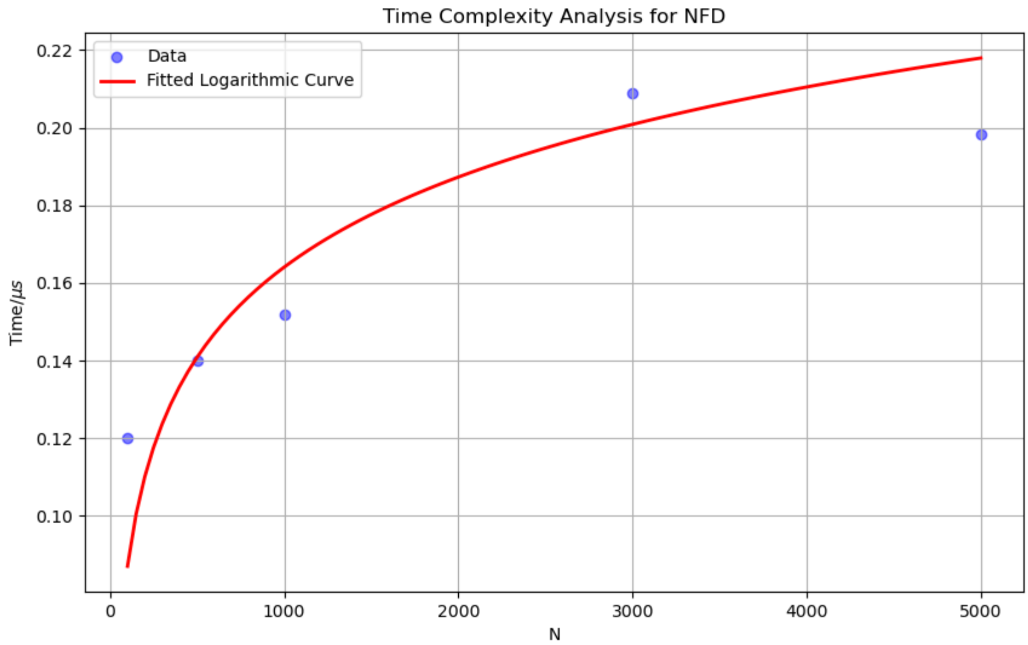

Figure 1: Performance of NFD

noted that the time complexity of NFD is $O(N\log(N))$, so the we make the curve of time/N versus N, a clearly log-like curve.

BFD:

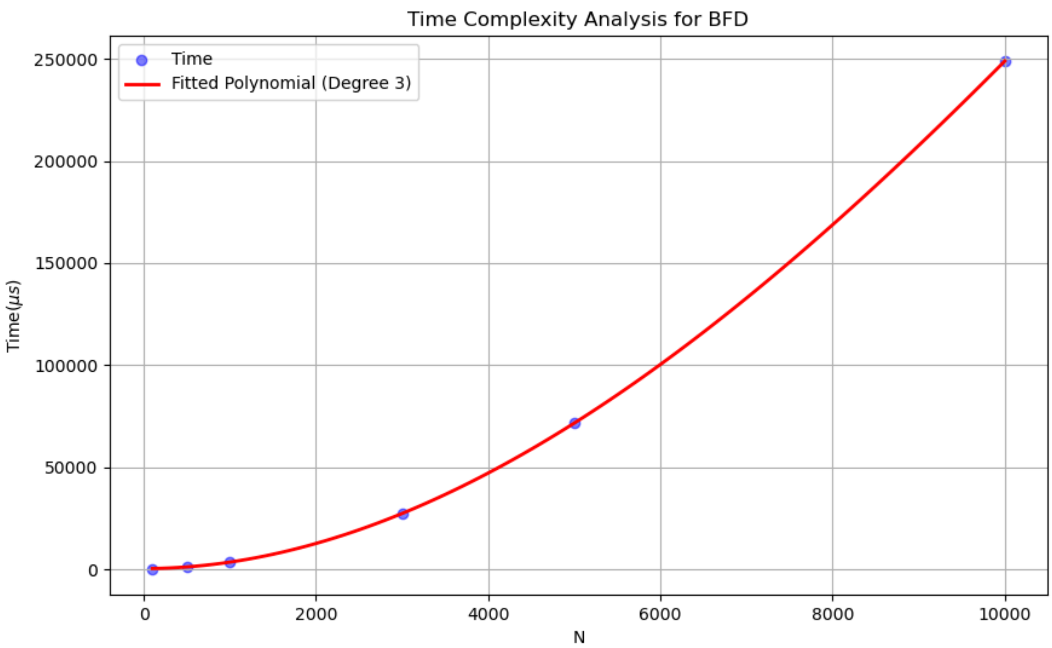

Figure 2: Performance of BFD

Chapter 4: Analysis and Comments

4.1) Time and Space Complexity Analysis

Space Complexity: all $O(N)$

Time Complexity: Already analyzed in Chapter 2

4.2) Approximation Comparison

Num	Optimal height	NF	NFD	BFD
5	371	797	583	583
100	12377	20253	16825	15028
500	62589	103175	81086	70037
1000	125882	209982	165349	144611
3000	376997	635886	499121	427661
5000	633510	1066305	842983	721865
10000	1243616	2105686	1658115	1409709

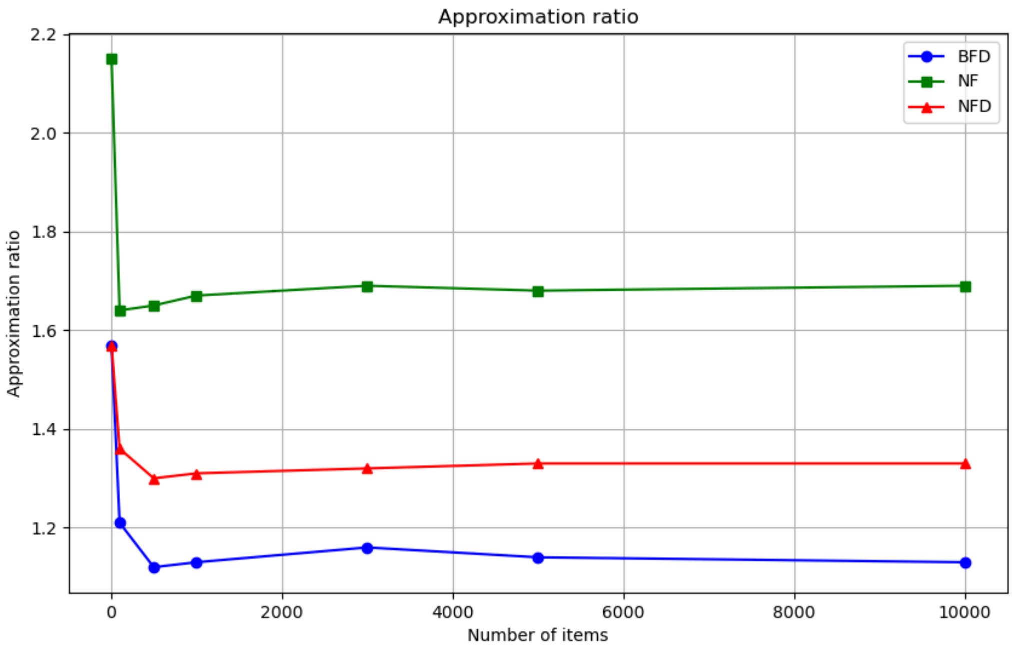

Figure 3: Performance of BFD

It is obvious that BFD outperforms the other two.

4.3) Factors that may affect approximation ratio:

4.3.1) Given Width:

We fix the number of items(100) and the optimal height(1000), and change the given width of the container.

Width	Approximation Ratio
50	1.101471
100	1.136792
200	1.085522
400	1.138715
500	1.124824
1000	1.135672
2000	1.152964
10000	1.158060
20000	1.314027
50000	1.580130

As we can see, the approximation ratio of BFD gets bigger as the given width increases. This is because when the width gets bigger, the average item gets bigger, leading to more space wasted on each level.

4.3.2) Ratio of width to height

Width/Height	Approximation Ratio
1:1	1.021471
1:2	1.104792
1:3	1.115522
1:4	1.131457
1:5	1.173124

As the ratio of width to height gets bigger, more space on each level is wasted because items are getting “fatter”, leading to higher approximation ratio.

4.4) Possible Improvements

The point is we need to rotate the rectangle when it is too fat or too slim in different circumstances, which can improve the ultilization of space.

There need to be a bound and a decisive function for when to rotate, we believe it is possible to compare four times when inserting an item in a block in BFDA, if the item or rotated item can be fit in the block, we insert it in.

Yet, this is still a not so perfect solution, therefore a better algorithm shall be implemented, and we still haven’t found a better one yet.

There are many algorithms done by computer scientists, like The split-fit algorithm (SF), Sleator’s algorithm, The split algorithm (SP), Reverse-fit (RF), and so on. Their approximation ratio is listed below.

Overview of polynomial time approximations

Year	Name	Approximation guarantee	Source
1980	Bottom-Up Left-Justified (BL)	$3OPT(I)$	Baker et al.[2]
1980	Next-Fit Decreasing-Height (NFDH)	$2OPT(I) + h_{\max}(I) \leq 3OPT(I)$	Coffman et al.[9]
	First-Fit Decreasing-Height (FFDH)	$1.7OPT(I) + h_{\max}(I) \leq 2.7OPT(I)$	
	Split-Fit (SF)	$1.5OPT(I) + 2h_{\max}(I)$	
1980		$2OPT(I) + h_{\max}(I)/2 \leq 2.5OPT(I)$	Sleator[10]
1981	Split Algorithm (SP)	$3OPT(I)$	Golan[11]
	Mixed Algoritghm	$(4/3)OPT(I) + 7\frac{1}{18}h_{\max}(I)$	
1981	Up-Down (UD)	$(5/4)OPT(I) + 6\frac{7}{8}h_{\max}(I)$	Baker et al.[12]
1994	Reverse-Fit	$2OPT(I)$	Schiermeyer[13]
1997		$2OPT(I)$	Steinberg[8]
2000		$(1+\varepsilon)OPT(I) + \mathcal{O}(1/\varepsilon^2)h_{\max}(I)$	Kenyon, Rémila[14]
2009		$1.9396OPT(I)$	Harren, van Stee[15]
2009		$(1+\varepsilon)OPT(I) + h_{\max}(I)$	Jansen, Solis-Oba[16]
2011		$(1+\varepsilon)OPT(I) + \mathcal{O}(\log(1/\varepsilon)/\varepsilon)h_{\max}(I)$	Bougeret et al.[17]
2012		$(1+\varepsilon)OPT(I) + \mathcal{O}(\log(1/\varepsilon)/\varepsilon)h_{\max}(I)$	Sviridenko[18]
2014		$(5/3+\varepsilon)OPT(I)$	Harren et al.[3]

Figure 4: Ratio

Appendix: Source Code (in C)

5.1) Next-Fit code

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>  //to calculate time


typedef struct
{
   int width;
   int height;
} item;  //define the structure of item
typedef item*  Item;


void downheap(Item* a, int i, int P)//its a maxheap implementation, maximum height
{
   while(2*i+1 <= P)//if the left child exist
   {
      int j = 2*i;
      if(j < P && a[j]->height < a[j+1]->height)//if the right child exist and the right child is larger than
```

```
the left child
          j++;
       if(a[i]->height >= a[j]->height) //if the parent is larger than the child
         break;
       Item temp = a[i];//swap the parent
       a[i] = a[j];
       a[j] = temp;
       i = j;
    }
}
void sortheap(Item* a, int P)//heap sort
{
    for (int i = P; i >= 1; i--)
      downheap(a, i, P);
}
void upheap(Item* a, int i)//upheap procedure
{
    while(i > 1 && a[i/2]->height < a[i]->height)//if the parent is smaller than the child
    {
       Item temp = a[i];//swap the parent
       a[i] = a[i/2];
       a[i/2] = temp;
       i = i/2;
    }
}
void insert(Item* a, Item x, int P)//insert a new item
{
    a[P + 1] = x;//insert the new item
    P++;
    upheap(a, P);//upheap the new item
}
void pop(Item* a, int P)//delete the top item
{
    if(P==0)//if the heap is empty
    {
       return;
    }
    a[1] = a[P];//swap the top item with the last item
    P--;
    downheap(a, 1, P);//downheap the top item
}
void deleteitem(Item* a, int i, int P)//delete the item at the ith position
{
    a[i] = a[P];
    P--;//swap the ith item with the last item
    downheap(a, i, P);
}
```

```
int main()//main function
{
  double start, end, result; //variables for time measurement

  int N, B, W;//introduce variables for input
  FILE* input;//file pointer
  input=fopen("test_cases/test.txt", "r");//open the input file
  fscanf(input,"%d %d", &N, &W);
  Item* item1 = (Item*)malloc((N+1)*sizeof(Item)); //allocate memory for the item array
  for(int i=0;i<=N;i++)//allocate memory for each item
  {
    item1[i]=(Item)malloc(sizeof(item));
  }
  for (int i = 1; i <= N; i++) //read the width and height of each item
    fscanf(input,"%d %d", &item1[i]->width, &item1[i]->height);
  start=clock();//the start of time

  Item* block = (Item*)malloc((2*N+1)*sizeof(Item));  //allocate memory for the block array
  for(int i=0;i<=N;i++)//allocate memory for each block
  {
    block[i]=(Item)malloc(sizeof(item));
  }
  block[1]->width=W-item1[1]->width;//initialize the first block
  block[1]->height=item1[1]->height;
  int H=item1[1]->height;
  int i=1;
  i++;
  N=N-1;
  int count=0;

  int assist1,assist2;
  //O(N)
  while(N>0)//when there are still items that aren't placed
  {

    if(block[1]->width>=item1[i]->width&&block[1]->height>=item1[i]->height)//if the item can be
placed in this block
    {
      assist1=block[1]->height;//store the height of the block
      assist2=block[1]->width;//store the width of the block

      block[1]->width=assist2-item1[i]->width;//update the width of the block
      block[1]->height=item1[i]->height;
      count++;

    }

    else//if the item cannot be placed in any block
```

```
        {
        block[1]->width=W-item1[i]->width;//create a new block
        block[1]->height=item1[i]->height;
        H=H+block[1]->height;  //update the height of the block

        count++;
        }
        i++;
        N=N-1;
    }


    end=clock();
    result=(end-start)*1000000/CLOCKS_PER_SEC;//calculate the time,and the unit is us
    printf("%d %lfus\n", H, result);


    free(item1);  //free the memory

    free(block);//free the memory
    free(input);//free the file

    return 0;
}
```

5.2) Next-Fit Decreasing code

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>  //to calculate time


typedef struct
{
    int width;
    int height;
} item;  //define the structure of item
typedef item*  Item;


void downheap(Item* a, int i, int P)//its a maxheap implementation, maximum height
{
    while(2*i+1 <= P)//if the left child exist
    {
        int j = 2*i;
        if(j < P && a[j]->height < a[j+1]->height)//if the right child exist and the right child is larger than
the left child
            j++;
```

```
    if(a[i]->height >= a[j]->height) //if the parent is larger than the child
      break;
    Item temp = a[i];//swap the parent
    a[i] = a[j];
    a[j] = temp;
    i = j;
  }
}
void sortheap(Item* a, int P)//heap sort
{
  for (int i = P; i >= 1; i--)
    downheap(a, i, P);
}
void upheap(Item* a, int i)//upheap procedure
{
  while(i > 1 && a[i/2]->height < a[i]->height)//if the parent is smaller than the child
  {
    Item temp = a[i];//swap the parent
    a[i] = a[i/2];
    a[i/2] = temp;
    i = i/2;
  }
}
void insert(Item* a, Item x, int P)//insert a new item
{
  a[P + 1] = x;//insert the new item
  P++;
  upheap(a, P);//upheap the new item
}
void pop(Item* a, int P)//delete the top item
{
  if(P==0)//if the heap is empty
  {
    return;
  }
  a[1] = a[P];//swap the top item with the last item
  P--;
  downheap(a, 1, P);//downheap the top item
}
void deleteitem(Item* a, int i, int P)//delete the item at the ith position
{
  a[i] = a[P];
  P--;//swap the ith item with the last item
  downheap(a, i, P);
}



int main()//main function
```

```
{
  double start, end, result; //variables for time measurement

  int N, B, W;//introduce variables for input
  FILE* input;//file pointer
  input=fopen("test_cases/test.txt", "r");//open the input file
  fscanf(input,"%d %d", &N, &W);
  Item* item1 = (Item*)malloc((N+1)*sizeof(Item)); //allocate memory for the item array
  for(int i=0;i<=N;i++)//allocate memory for each item
  {
    item1[i]=(Item)malloc(sizeof(item));
  }
  for (int i = 1; i <= N; i++) //read the width and height of each item
    fscanf(input,"%d %d", &item1[i]->width, &item1[i]->height);
  start=clock();//the start of time
  sortheap(item1, N);  //heap sort
  Item* block = (Item*)malloc((2*N+1)*sizeof(Item));  //allocate memory for the block array
  for(int i=0;i<=N;i++)//allocate memory for each block
  {
    block[i]=(Item)malloc(sizeof(item));
  }
  block[1]->width=W-item1[1]->width;//initialize the first block
  block[1]->height=item1[1]->height;
  int H=item1[1]->height;

  pop(item1, N);//delete the first item
  N=N-1;
  int count=0;

  int assist1,assist2;
  //O(Nlog(N))
  while(N>0)//when there are still items that aren't placed
  {

    if(block[1]->width>=item1[1]->width&&block[1]->height>=item1[1]->height)//if the item can be placed in this block
    {
      assist1=block[1]->height;//store the height of the block
      assist2=block[1]->width;//store the width of the block

      block[1]->width=assist2-item1[1]->width;//update the width of the block
      block[1]->height=item1[1]->height;
      count++;

    }

    else//if the item cannot be placed in any block
    {
    block[1]->width=W-item1[1]->width;//create a new block
```

```
        block[1]->height=item1[1]->height;
        H=H+block[1]->height;  //update the height of the block

        count++;
        }
        pop(item1, N);//delete the item
        N=N-1;
    }


    end=clock();
    result=(end-start)*1000000/CLOCKS_PER_SEC;//calculate the time,and the unit is us
    printf("%d %lfus\n", H, result);


    free(item1);  //free the memory

    free(block);//free the memory
    free(input);//free the file

    return 0;
}
```

5.3) Best-Fit Decreasing Advanced code

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>  //to calculate time


typedef struct
{
    int width;
    int height;
} item;  //define the structure of item
typedef item*  Item;


void downheap(Item* a, int i, int P)//its a maxheap implementation, maximum height
{
    while(2*i+1 <= P)//if the left child exist
    {
        int j = 2*i;
        if(j < P && a[j]->height < a[j+1]->height)//if the right child exist and the right child is larger than
the left child
            j++;
        if(a[i]->height >= a[j]->height) //if the parent is larger than the child
          break;
```

```
        Item temp = a[i];//swap the parent
        a[i] = a[j];
        a[j] = temp;
        i = j;
    }
}
void sortheap(Item* a, int P)//heap sort
{
    for (int i = P; i >= 1; i--)
      downheap(a, i, P);
}
void upheap(Item* a, int i)//upheap procedure
{
    while(i > 1 && a[i/2]->height < a[i]->height)//if the parent is smaller than the child
    {
        Item temp = a[i];//swap the parent
        a[i] = a[i/2];
        a[i/2] = temp;
        i = i/2;
    }
}
void insert(Item* a, Item x, int P)//insert a new item
{
    a[P + 1] = x;//insert the new item
    P++;
    upheap(a, P);//upheap the new item
}
void pop(Item* a, int P)//delete the top item
{
    if(P==0)//if the heap is empty
    {
        return;
    }
    a[1] = a[P];//swap the top item with the last item
    P--;
    downheap(a, 1, P);//downheap the top item
}
void deleteitem(Item* a, int i, int P)//delete the item at the ith position
{
    a[i] = a[P];
    P--;//swap the ith item with the last item
    downheap(a, i, P);
}
void delete1(Item* a, int i, int P)
{
    for(int j=i;j<P;j++)
    {
        a[j]=a[j+1];
    }
```

```
    P--;
}
void insertitem(Item* a, int P)
{
    for(int i=P;i>1;i--)
    {
        if(a[i]->height>a[i+1]->height)
        {
            Item temp=a[i];
            a[i]=a[i+1];
            a[i+1]=temp;
        }
        else
        {
            break;
        }
    }
}


int main()//main function
{
    double start, end, result; //variables for time measurement

    int N, B, W;//introduce variables for input
    FILE* input;//file pointer
    input=fopen("test_cases/test.txt", "r");//open the input file
    fscanf(input,"%d %d", &N, &W);
    Item* item1 = (Item*)malloc((N+1)*sizeof(Item)); //allocate memory for the item array
    for(int i=0;i<=N;i++)//allocate memory for each item
    {
        item1[i]=(Item)malloc(sizeof(item));
    }
    for (int i = 1; i <= N; i++) //read the width and height of each item
        fscanf(input,"%d %d", &item1[i]->width, &item1[i]->height);
    start=clock();//the start of time
    sortheap(item1, N);  //heap sort
    Item* block = (Item*)malloc((2*N+1)*sizeof(Item));  //allocate memory for the block array
    for(int i=0;i<=N;i++)//allocate memory for each block
    {
        block[i]=(Item)malloc(sizeof(item));
    }
    block[1]->width=W-item1[1]->width;//initialize the first block
    block[1]->height=item1[1]->height;
    int H=item1[1]->height;
    int i;
    pop(item1, N);//delete the first item
    N=N-1;
    int count=0;
```

```
    int amountBlock=1;//initialize the amount of block
    int assist1,assist2;

    while(N>0)//when there are still items that aren't placed $O(N)~O(N^3)$
    {
    for(i=1;i<=amountBlock;i++)//when there are still blocks that haven't been chosen
    {
       if(block[i]->width>=item1[1]->width&&block[i]->height>=item1[1]->height)//if the item can be
placed in this block
       {
          assist1=block[i]->height;//store the height of the block
          assist2=block[i]->width;//store the width of the block
          delete1(block, i ,amountBlock); //delete the item
          amountBlock--;
          block[amountBlock+1]->width=assist2-item1[1]->width;//update the width of the block
          block[amountBlock+1]->height=item1[1]->height;

          insertitem(block, amountBlock);
          amountBlock++; //insert the new block
          block[amountBlock+1]->width=item1[1]->width;  //create a new block
          block[amountBlock+1]->height=assist1-item1[1]->height;  //update the height of the block
          insertitem(block, amountBlock);
          amountBlock++;  //insert the new block

          count++;
          break;
       }
    }
    if(i>amountBlock)//if the item cannot be placed in any block
    {
       block[amountBlock+1]->width=W-item1[1]->width;//create a new block
       block[amountBlock+1]->height=item1[1]->height;
       H=H+block[amountBlock+1]->height;  //update the height of the block
       insertitem(block,amountBlock);
       amountBlock++;
       count++;
    }
    pop(item1, N);//delete the item
    N=N-1;
    }
    end=clock();
    result=(end-start)*1000000/CLOCKS_PER_SEC;//calculate the time,and the unit is us
    printf("%d %lfus\n", H, result);


    free(item1);  //free the memory

    free(block);//free the memory
    free(input);//free the file
```

```
    return 0;
}
```

Declaration

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