

# 项目说明文档

## 数据结构课程设计

——修理牧场

作者姓名： 翟晨昊

学 号： 1952216

指导教师： 张颖

学院、专业： 软件学院 软件工程

同济大学

Tongji University

## 目 录

1	分析.....	- 3 -
1.1	背景分析 .....	- 3 -
1.2	功能分析 .....	- 3 -
2	设计.....	- 4 -
2.1	数据结构设计.....	- 4 -
2.2	类结构设计 .....	- 4 -
2.3	成员与操作设计 .....	- 4 -
2.4	系统设计 .....	- 8 -
3	实现.....	- 9 -
3.1	计算最小花费功能的实现 .....	- 9 -
3.1.1	计算最小花费功能流程图.....	- 9 -
3.1.2	计算最小花费功能核心代码.....	- 10 -
3.1.3	计算最小花费功能描述.....	- 12 -
3.2	总体功能的实现.....	- 13 -
3.2.1	总体功能流程图.....	- 13 -
3.2.2	总体功能核心代码 .....	- 14 -
3.2.3	总体功能截屏示例 .....	- 15 -
4	测试.....	- 16 -
4.1	功能测试 .....	- 16 -
4.1.1	基本功能测试.....	- 16 -
4.1.2	木块数量为一 .....	- 16 -
4.1.3	每块木头长度相同 .....	- 16 -
4.2	边界测试 .....	- 18 -
4.2.1	木块数量为零.....	- 18 -
4.2.2	木块数量大于要求最大值.....	- 18 -
4.2.3	存在木块长度为零 .....	- 18 -
4.3	出错测试 .....	- 19 -
4.3.1	输入木块长度个数比木块总数多 .....	- 19 -
4.3.2	输入木块个数错误 .....	- 19 -
4.3.3	木块长度输入不合法 .....	- 20 -

# 1 分析

## 1.1 背景分析

农夫要修理牧场的一段栅栏，他测量了栅栏，发现需要  $N$  块木头，每块木头长度为整数  $L_i$  个长度单位，于是他购买了一个很长的，能锯成  $N$  块的木头，即该木头的长度是  $L_i$  的总和。

但是农夫自己没有锯子，请人锯木的酬金跟这段木头的长度成正比。为简单起见，不妨就设酬金等于所锯木头的长度。例如，要将长度为 20 的木头锯成长度为 8，7 和 5 的三段，第一次锯木头将木头锯成 12 和 8，花费 20；第二次锯木头将长度为 12 的木头锯成 7 和 5 花费 12，总花费 32 元。如果第一次将木头锯成 15 和 5，则第二次将木头锯成 7 和 8，那么总的花费是 35（大于 32）。

## 1.2 功能分析

本程序的目的是为了计算出将木头锯成目标块数和长度所需要的最少花费。首先，程序要够储存用户的需求，即要把木头锯成几块与每块木头的长度。随后，程序通过计算，得出所需要的最少花费，该花费是一个整数。最后将这个整数输出。

综上所述，本程序中需要有输入，计算最小花费，输出的功能。

## 2 设计

### 2.1 数据结构设计

如上功能分析所述，每次锯木头的酬金等于所锯木头的长度，为了让系统能输出最小花费，就要尽量避免对长的木头进行多次操作。输入已经给出了每段木头要切成的长度，我们可以通过将切下的木头还原来分析整个过程。例如一根木头被锯成十段，每段的长度分别为 1 2 3 4 5 6 7 8 9 10。由于我们要避免对长的木头进行多次操作，因此第一次我们将最短的两根木头拼在一起， $1+2=3$ ，现在每段的长度分别为 3 3 4 5 6 7 8 9 10，以此类推，再复原  $3+3=6$ ， $4+5=9$ ……每次的酬金都为当前可能的最小值，最后得出的总酬金就会最少。

通过上面的分析，我们可以用一个数组来存储每段长度，每次从中取出最小的两个值加在一起在放回数组，直至数组中只剩下一个值；也可以设计一个最小堆，每次从堆顶相继取出两个长度，再把新生成的长度加入堆中；最后也可以构建一个霍夫曼树来解决问题。本项目最终选择了最小堆数据结构来实现要求。

### 2.2 类结构设计

首先，我们需要一个堆类（Heap 类），用来在过程中对每段木头的长度进行排序。由于木头的数量未知，因此采用 Vector 向量类来储存每段木头的长度。为了保证数据结构的泛用性，我们把堆与 Vector 类都设计为了模板类，同时，设计了两个 function object，分别为 Greater 类与 Less 类，来帮助 Heap 类实现最大堆与最小堆。本项目中需要的是最小堆。

### 2.3 成员与操作设计

向量类（Vector）：

```
template <typename ElementType> class Vector {
public:
    ~Vector();
    Vector();
    ElementType& operator[](const int x);
    const ElementType& operator[](const int x)const;
    Vector<ElementType>(const Vector<ElementType>& rhs);
    Vector<ElementType>& operator=(const Vector<ElementType>& rhs);
    bool isFull();
    bool isEmpty();
    void pushBack(const ElementType& temp);
    void popBack();
    void clear();
    int getSize();
    void reSize(int newSize);
private:
```

```
void extendSize();  
int size;  
int maxSize;  
ElementType* myVector;  
};
```

**私有成员:**

int size;//Vector 中已经存储的元素数量  
int maxSize;//Vector 中已经申请的空间大小，元素数量如果超过要再次申请  
ElementType\* myVector;//存储的元素序列的起始地址

**私有操作:**

```
void extendSize();  
//在 Vector 申请的空间已经被占满时再次申请空间，每次扩充至 maxsize*2+1  
是因为刚开始的 maxsize 为 0
```

**公有操作:**

```
Vector();  
//构造函数，初始化指针并将 size 与 maxSize 置为 0
```

```
~Vector();  
//析构函数，调用 clear() 函数来删除元素，释放内存
```

```
ElementType& operator[] (const int x);  
//重载[]运算符，返回 ElementType&类型
```

```
const ElementType& operator[] (const int x) const;  
//重载[]运算符，返回 const ElementType&类型
```

```
Vector<ElementType>(const Vector<ElementType>& rhs);  
//复制构造函数，将一个 Vector 复制给另一个 Vector
```

```
Vector<ElementType>& operator=(const Vector<ElementType>& rhs);  
//重载=运算符，可以将一个 Vector 赋给另一个 Vector
```

```
bool isFull();  
//判断是否 Vector 中申请的内存已经被占满
```

```
bool isEmpty();  
//判断 Vector 是否为空
```

```
void pushBack(const ElementType& temp);  
//在 Vector 末尾添加一个元素
```

```
void popBack();  
//删除 Vector 最末尾的元素  
  
void clear();  
//删除 Vector 中的所有元素并释放内存  
  
int getSize();  
//返回 Vector 已经存储的元素数量  
  
void reSize(int newSize);  
//重设 Vector 的大小，若比之前小，则会抛弃多余的元素
```

**堆类 (Heap):**

```
template<typename ElementType, typename Comparator> class Heap{  
public:  
    Heap() = default;  
    ~Heap();  
    int getSize();  
    void build(Vector<ElementType>& everyLength);  
    void insert(const ElementType& inputData);  
    bool pop();  
    ElementType getTop();  
    void makeEmpty();  
private:  
    Vector<ElementType> data;  
    void siftDown(int start, int max);  
    void siftUp(int start);  
};
```

**私有成员:**

Vector<ElementType> data; //保存堆中的数据

**私有操作:**

```
void siftDown(int start, int max);  
//从结点 start 到结点 max 为止下滑调整堆中数据
```

```
void siftUp(int start);  
//从结点 start 到最上方上滑调整堆中数据
```

**公有操作:**

```
Heap() = default;  
//默认构造函数
```

```
~Heap();  
//析构函数，调用 makeEmpty() 函数来删除元素，释放内存
```

```
int getSize();  
//返回堆已经存储的元素数量
```

```
void build(Vector<ElementType>& everyLength);  
//通过 Vector 构建堆
```

```
void insert(const ElementType& inputData);  
//向堆中插入一个元素
```

```
bool pop();  
//删除堆中的一个元素
```

```
ElementType getTop();  
//返回堆顶元素
```

```
void makeEmpty();  
//删除堆中的所有元素并释放内存
```

**大于比较类 (Greater):**

```
template<typename ElementType> class Greater{  
public:  
    bool operator()(const ElementType& E1, const ElementType& E2)  
    {  
        return E1 > E2;  
    }  
};
```

**公有操作:**

```
bool operator()(const ElementType& E1, const ElementType& E2);  
//当 E1>E2 时返回 true
```

**小于比较类 (Less):**

```
template<typename ElementType> class Less{  
public:  
    bool operator()(const ElementType& E1, const ElementType& E2)  
    {  
        return E1 < E2;  
    }  
};
```

公有操作:

```
bool operator() (const ElementType& E1, const ElementType& E2);  
//当 E1<E2 时返回 true
```

## 2.4 系统设计

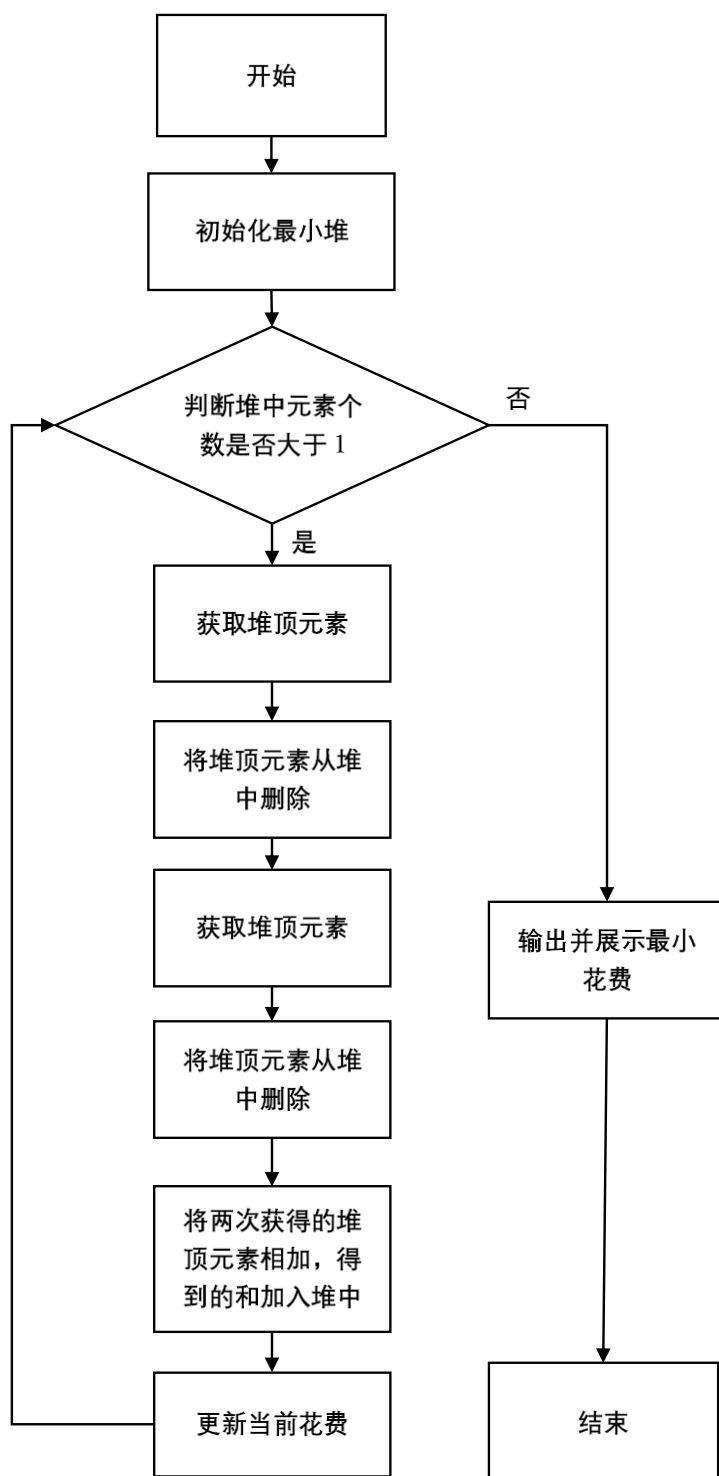
程序在使用时，首先输入木块数量（不超过 10000），随后在输入每块木头的长度，将长度序列加入堆中并排序，计算所需的最小花费后，将其输出并显示。



### 3 实现

#### 3.1 计算最小花费功能的实现

##### 3.1.1 计算最小花费功能流程图



## 3.1.2 计算最小花费功能核心代码

```

void calculate(Vector<int>& everyLength)
{
    Heap<int, Greater<int> > minHeap;
    int sum = 0;
    int newLength = 0;
    minHeap.build(everyLength);
    while (minHeap.getSize() > 1)
    {
        int length1 = minHeap.getTop();
        if (!minHeap.pop())
        {
            cout << "删除失败! ";
            return;
        }
        int length2 = minHeap.getTop();
        if (!minHeap.pop())
        {
            cout << "删除失败! ";
            return;
        }
        newLength = length1 + length2;
        sum += newLength;
        minHeap.insert(newLength);
    }
    cout << "最小花费为: " << sum << endl;
}

```

Heap 类中:

```

template<typename ElementType, typename Comparator>
void Heap<ElementType, typename Comparator>::build(Vector<ElementType>&
    everyLength)
{
    for (int i = 0; i < everyLength.getSize(); i++)
    {
        data.pushBack(everyLength[i]);
    }
    int currentPos = (getSize() - 2) / 2;
    while (currentPos >= 0)
    {
        siftDown(currentPos, getSize() - 1);
        currentPos--;
    }
}

```

```
}

template<typename ElementType, typename Comparator>
void Heap<ElementType, typename Comparator>::insert(const ElementType&
inputData)
{
    data.pushBack(inputData);
    siftUp(getSize() - 1);
}

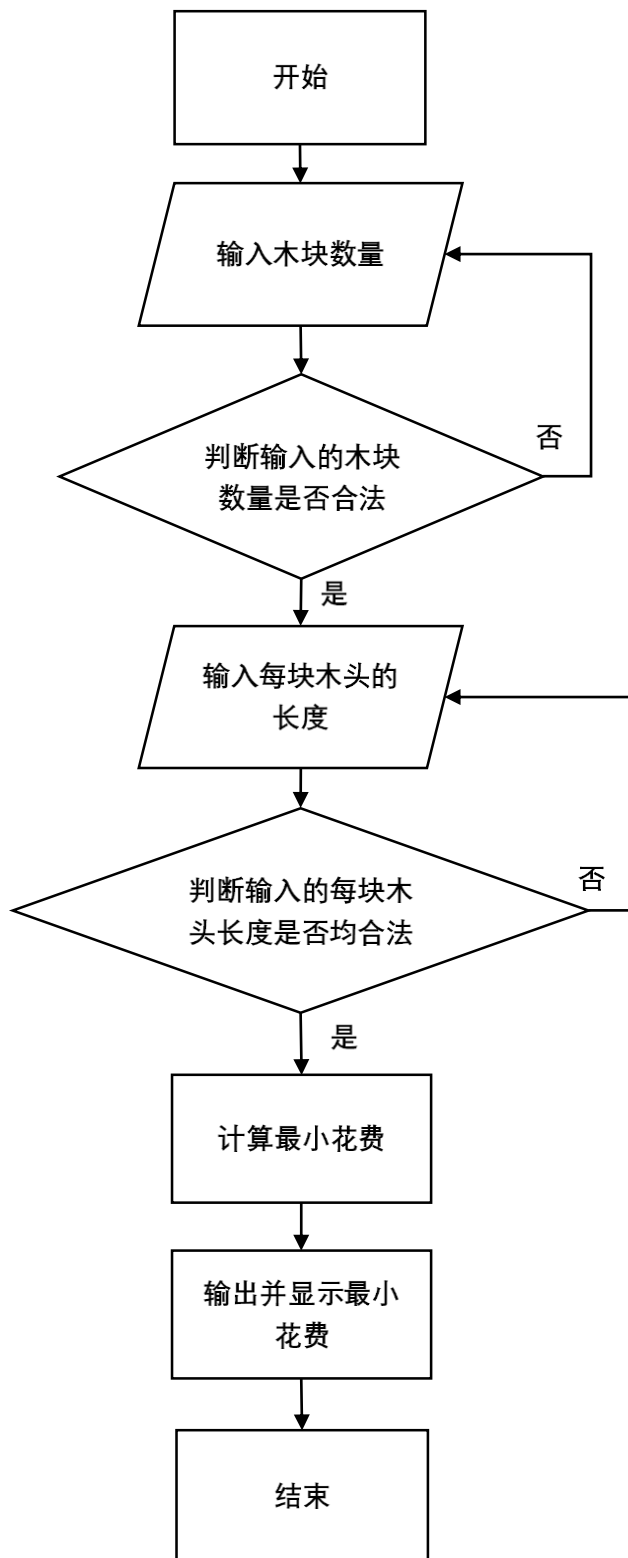
template<typename ElementType, typename Comparator>
bool Heap<ElementType, typename Comparator>::pop()
{
    if (getSize() == 0)
    {
        return false;
    }
    data[0] = data[getSize() - 1];
    data.popBack();
    siftDown(0, getSize() - 1);
    return true;
}
```

### 3.1.3 计算最小花费功能描述

刚开始要根据用户输入的木块个数与每块木块长度，来初始化一个最小堆，随后开始计算。如果堆中元素大于 1，就依次从堆顶取出堆中最小的两个元素，将它们相加，就是这一次的花费。将它们相加的和重新加入堆中，并且维护这个堆，使其仍然是一个最小堆。同时把它们相加的和计入花费之中。当堆中元素唯一时，就可以输出花费，此花费即为最小花费。

## 3.2 总体功能的实现

### 3.2.1 总体功能流程图



### 3.2.2 总体功能核心代码

```
int main()
{
    Vector<int> everyLength;
    input(everyLength);
    calculate(everyLength);
    return 0;
}

void input(Vector<int>& everyLength)
{
    int num = 0;
    int inputLength = 0;
    cout << "请输入木块数量: " << endl;
    cin >> num;
    while (cin.fail() || num <= 0 || num > 10000)
    {
        cout << "木块数量只能是正整数且不能超过 10000!" << endl;
        cout << "请重新输入数量: " << endl;
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cin >> num;
    }
    cout << "请输入每块木头的长度: " << endl;
    for (int i = 0; i < num; i++)
    {
        cin >> inputLength;
        if (cin.fail() || inputLength <= 0)
        {
            cout << "木块长度只能是正整数!" << endl;
            cout << "请重新输入长度: " << endl;
            everyLength.clear();
            i = 0;
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            cin >> inputLength;
        }
        everyLength.pushBack(inputLength);
    }
}
```

### 3.2.3 总体功能截屏示例

```
请输入木块数量:  
8  
请输入每块木头的长度:  
4 5 1 2 1 3 1 1  
最小花费为: 49
```

## 4 测试

### 4.1 功能测试

#### 4.1.1 基本功能测试

测试用例：

8

1 2 3 4 5 6 7 8

预期结果：102

实验结果：

```
请输入木块数量：
8
请输入每块木头的长度：
1 2 3 4 5 6 7 8
最小花费为：102
```

#### 4.1.2 木块数量为一

测试用例：

1

15

预期结果：0

实验结果：

```
请输入木块数量：
1
请输入每块木头的长度：
15
最小花费为：0
```

#### 4.1.3 每块木头长度相同

测试用例：

10

1 1 1 1 1 1 1 1 1 1

预期结果：0



实验结果:

```
请输入木块数量:
10
请输入每块木头的长度:
1 1 1 1 1 1 1 1 1 1
最小花费为: 34
```

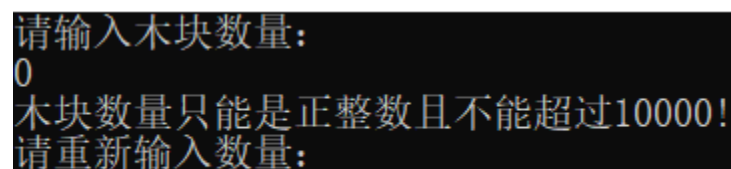
## 4.2 边界测试

### 4.2.1 木块数量为零

测试用例：0

预期结果：程序给出提示信息，程序正常运行不崩溃。

实验结果：



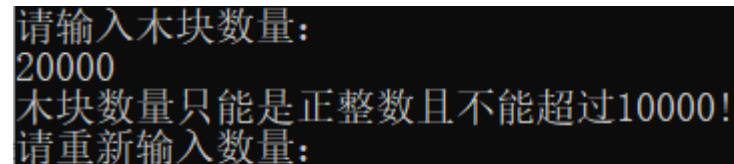
```
请输入木块数量：
0
木块数量只能是正整数且不能超过10000!
请重新输入数量：
```

### 4.2.2 木块数量大于要求最大值

测试用例：20000

预期结果：程序给出提示信息，程序正常运行不崩溃。

实验结果：



```
请输入木块数量：
20000
木块数量只能是正整数且不能超过10000!
请重新输入数量：
```

### 4.2.3 存在木块长度为零

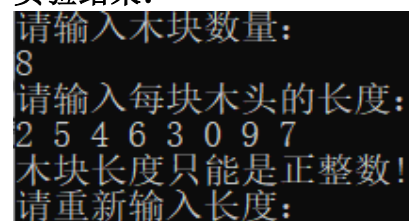
测试用例：

8

2 5 4 6 3 0 9 7

预期结果：程序给出提示信息，程序正常运行不崩溃。

实验结果：



```
请输入木块数量：
8
请输入每块木头的长度：
2 5 4 6 3 0 9 7
木块长度只能是正整数!
请重新输入长度：
```

## 4.3 出错测试

### 4.3.1 输入木块长度个数比木块总数多

测试用例:

8

1 2 3 4 5 6 7 8 9 10

预期结果: 程序忽略多出的木块长度 (9 10), 输出 102

实验结果:

```
请输入木块数量:
8
请输入每块木头的长度:
1 2 3 4 5 6 7 8 9 10
最小花费为: 102
```

### 4.3.2 输入木块个数错误

测试用例:

-5

wasd

测试

预期结果: 程序给出提示信息, 程序正常运行不崩溃。

实验结果:

```
请输入木块数量:
-5
木块数量只能是正整数且不能超过10000!
请重新输入数量:
wasd
木块数量只能是正整数且不能超过10000!
请重新输入数量:
测试
木块数量只能是正整数且不能超过10000!
请重新输入数量:
```

### 4.3.3 木块长度输入不合法

测试用例：

8

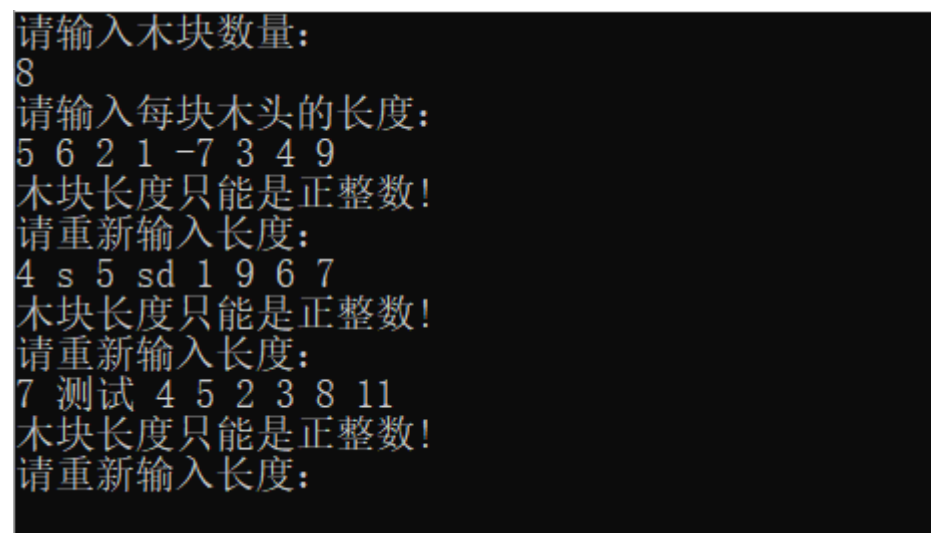
5 6 2 1 -7 3 4 9

4 s 5 sd 1 9 6 7

7 测试 4 5 2 3 8 11

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**



```
请输入木块数量：
8
请输入每块木头的长度：
5 6 2 1 -7 3 4 9
木块长度只能是正整数！
请重新输入长度：
4 s 5 sd 1 9 6 7
木块长度只能是正整数！
请重新输入长度：
7 测试 4 5 2 3 8 11
木块长度只能是正整数！
请重新输入长度：
```