法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

关注 小象学院

第七课　　指针与函数进阶

林沐

内容概述

1.函数指针引入
2.函数指针数组
3.函数指针用作函数参数
4.例1-通用的选择排序
5.函数中的静态变量
6.函数间共享变量:全局变量
7.main函数
8.参数数量可变的函数
9.例2-简单版本的printf函数
10.C语言库函数:快速排序qsort
11.C语言库函数:二分查找bsearch
12.例3-火柴杆摆成方形

函数指针

函数的**内存地址**存储了函数**开始执行**的位置，它保存在**函数名**中(可以将函数名当作一个常量看待)。使用**指针**保存函数的地址，即使指针**指向**这个函数，指向函数的指针被称作是**函数指针**。

通过函数指针可以**灵活的**调用各种**形式相同**(函数返回值与函数参数列表相同)但是**功能不同**的函数，大大增加了代码的**灵活程度**。

```
p_func sizeof = 4
sum's address = 004012F0
p_func = 004012F0
sum = 13
difference = -3
product = 40
请按任意键继续. . .
```

函数指针声明: 三个要素　　函数返回值 int　　函数指针名 (*p_func)　　函数参数列表 (int, int)

```
#include <stdio.h>
int sum(int x, int y){
    return x + y;
}
int difference(int x, int y){
    return x - y;
}
int product(int x, int y){
    return x * y;
}

int main(){
    int a = 5;
    int b = 8;
    int (*p_func)(int, int);
    printf("p_func sizeof = %d\n", sizeof(p_func));
    p_func = sum;
    //打印函数的地址
    printf("sum's address = %p\n", sum);
    printf("p_func = %p\n", p_func);
    //通过p_func调用sum函数，返回a与b的和
    printf("sum = %d\n", p_func(a, b));
    p_func = difference;
    printf("difference = %d\n", p_func(a, b));
    p_func = product;
    printf("product = %d\n", p_func(a, b));

    return 0;
}
```

//这三个函数的形式都是一样的:
包括相同的函数返回值与参数列表

//声明了一个名叫p_func的，返回值为int，接收两个整型参数的函数指针

//将p_func指向sum函数

函数指针数组

若需要使用一组函数指针，则可定义**函数指针数组**。函数指针数组与指针数组或者是普通变量的数组没有本质区别，在**初始化**的时候可以使用**相同的方式**，它就是记录**函数地址**、**指向函数**的一组指针。

```c
#include <stdio.h>

int sum(int x, int y){
    return x + y;
}

int difference(int x, int y){
    return x - y;
}

int product(int x, int y){
    return x * y;
}
```

```c
int main(){
    int a = 5;
    int b = 8;

    int (*p_func[3])(int, int) = {
        sum,
        difference,
        product
    };

    char name[3][20] = {
        "sum",
        "difference",
        "product"
    };

    printf("p_func sizeof = %d\n", sizeof(p_func));

    int i;
    for (i = 0; i < 3; i++){
        printf("%s(%d, %d) = %d\n",
            name[i], a, b, p_func[i](a, b));
    }
    return 0;
}
```

//定义一个3个长度的指针数组
并赋初值，分别指向三个函数

//打印函数指针数组的长度

//用循环遍历函数指针数组，并调用

```
p_func sizeof = 12
sum(5, 8) = 13
difference(5, 8) = -3
product(5, 8) = 40
请按任意键继续. . .
```

函数指针用作函数参数

我们可以将**函数指针**作为**函数参数**传递给函数，这样函数就可以根据指针所指向的**函数不同**而调用不同的函数了。在这个函数中，通过该函数指针调用的函数被称为**回调函数**，这种开发方式的用途非常之广泛。

回调函数，就是**某函数**的**使用者**定义一个函数，使用者实现这个函数的程序内容，然后把这个**函数作为参数**传入该函数中，该函数在运行时通过**函数指针**调用的函数。换句话说，就是在别人写的函数的**运行期间**来**回调**你实现的函数。

```
sum = 13
difference = -3
product = 40
请按任意键继续. . .
```

```c
#include <stdio.h>

int sum(int x, int y){
    return x + y;
}                                   //你写的

int difference(int x, int y){
    return x - y;
}

int product(int x, int y){
    return x * y;
}
```

//这三个一般是使用者写的函数，被称为回调函数

//这是原函数，一般是开发者写的实现了很复杂功能的函数

```c
int compute_func(int (*p_func)(int, int), int x, int y){
    return p_func(x, y);            //别人写的
}

int main(){
    int a = 5;
    int b = 8;
    int result;

    result = compute_func(sum, a, b);
    printf("sum = %d\n", result);
    result = compute_func(difference, a, b);
    printf("difference = %d\n", result);
    result = compute_func(product, a, b);
    printf("product = %d\n", result);

    return 0;
}
```

//该函数在运行期间通过函数指针p_func调用了使用者开发的自定义某种功能的函数

//compute_func在运行时，回调了sum函数

例1-通用的选择排序

实现一个可以对**任何数据类型**的数组进行排序并且**排序规则**可以由**使用者定义**的**选择排序函数**。这个函数不仅可以排序**int、double、字符串、字符串指针**等数据类型的数组(还包括第8课学习的结构体数组)，还可以由用户指定对数组进行**升序排序**还是**降序排序**。

```
int int_array[8] = {2, 50, 3, -7, 2, 9, 0, 0};  //长度为8的整型数组

//长度为8的双精度数组
double double_array[8] = {2.5, 13.12, -0.7, 0, -50, 9, -0.123, 0};

const char *str_array_ptr[8] = {
    "zzz",
    "zzzz",
    "xxxxxxxxx",
    "p",
    "abc",
    "ccckkk",
    "aaa",
    "mmwordilovecoding"
};
//长度为8的字符串指针数组

char str_array[8][100] = {
    "zzz",
    "zzzz",
    "xxxxxxxxx",
    "p",
    "abc",
    "ccckkk",
    "aaa",
    "mmwordilovecoding"
};
//长度为8的字符串数组(二维字符数组)
```

思考:

1.如何设计函数，才可以使得排序函数可以对**任意数据类型**的数组进行排序?

2.如何设计函数，才可以使得排序函数可以根据函数**使用者的指定**对数组升序或降序排序?

例1-思考与分析

函数的原型设计:

void sort_array(void *base, int num, int width,
int (*compare)(const void *, const void *))

void *base : 指向**待排序**的数组。

int num: 待排序数组的**元素个数**。

int width: 待排序数组的**元素大小**。

int (*compare)(const void *, const void *):

排序时的**回调函数**，用户传进来一个回调函数，在比较两元素的时候，使用这个回调函数对元素进行比较，就可以实现自定义的**升序排列**或**降序排列**了。

思考:

1.由于只传进来数组的**首地址**，如何根据数组首地址**访问各个元素**?这时还需要利用函数原型中的**哪个参数**?

2.在访问每个元素时，**回调函数**应如何设计，就可以**比较**每个元素了?**升序排列**和**降序排列**的回调函数有何不同?(举例按升序**排序整型数组**时回调函数的写法)

3.当需要**交换**数组中两个元素的位置时，在不知道元素的类型时，应**如何交换**这两个元素?

例1-调用程序

```
//智能的排序函数，传入参数依次是待排序数组地址、
      数组元素个数、元素大小、用户的回调函数指针
void sort_array(void *base, int num, int width,
    int (*compare)(const void *, const void *)){
}
        //比较整型的回调函数
int int_cmp(const void *a, const void *b){
}
        //比较浮点型的回调函数
int double_cmp(const void *a, const void *b){
}
        //比较字符串数组的回调函数
int str_cmp(const void *a, const void *b){
}
        //比较字符串指针数组的回调函数
int str_ptr_cmp(const void *a, const void *b){
}

int main(){
    int int_array[8] = {2, 50, 3, -7, 2, 9, 0, 0};
    double double_array[8] =
         {2.5, 13.12, -0.7, 0, -50, 9, -0.123, 0};
    const char *str_array_ptr[8] = {
        "zzz",
        "zzzz",
        "xxxxxxxxx",
        "p",
        "abc",
        "ccckkk",
        "aaa",
        "mmwordilovecoding"
    };
    char str_array[8][100] = {
        "zzz",
        "zzzz",
        "xxxxxxxxx",
        "p",
        "abc",
        "ccckkk",
        "aaa",
        "mmwordilovecoding"
    };
    int i;
    sort_array(int_array, 8, sizeof(int), int_cmp);
    printf("int_array:\n");
    for (i = 0; i < 8; i++){
        printf("%d ", int_array[i]);
    }
    printf("\n\n");
    sort_array(double_array, 8, sizeof(double), double_cmp);
    printf("double_array:\n");
    for (i = 0; i < 8; i++){
        printf("%lf ", double_array[i]);
    }
    printf("\n\n");

    sort_array(str_array_ptr, 8, sizeof(str_array_ptr[0]), str_ptr_cmp);
    printf("str_array_ptr:\n");
    for (i = 0; i < 8; i++){
        printf("%s\n", str_array_ptr[i]);
    }
    printf("\n");

    sort_array(str_array, 8, sizeof(str_array[0]), str_cmp);
    printf("str_array:\n");
    for (i = 0; i < 8; i++){
        printf("%s\n", str_array[i]);
    }
    return 0;
}
```

```
int_array:
-7 0 0 2 2 3 9 50

double_array:
-50.000000 -0.700000 -0.123000 0.000000
 0.000000 2.500000 9.000000 13.120000
str_array_ptr:
aaa
abc
ccckkk
mmwordilovecoding
p
xxxxxxxxx
zzz
zzzz

str_array:
aaa
abc
ccckkk
mmwordilovecoding
p
xxxxxxxxx
zzz
zzzz
请按任意键继续. . .
```

例1-遍历传入的数组

```c
#include <stdio.h>
//遍历并打印任意数据类型的数组元素
void print_array(void *base, int num, int width,
                 void (*print)(const void *)){
    int i;
    for (i = 0; i < num; i++){ //计算每个元素
        void *element = base + i * width;
        print(element); //回调
    }
    printf("\n");
}                          //回调函数

void int_print(const void *a){
    printf("%d ", *(int *)a);
}

void str_print(const void *a){
    printf("%s\n", (char *)a);
}
```

```c
int main(){
    int int_array[8] =
        {2, 50, 3, -7, 2, 9, 0, 0};
    char str_array[8][100] = {
        "zzz",
        "zzzz",
        "xxxxxxxxx",
        "p",
        "abc",
        "ccckkk",
        "aaa",
        "mmwordilovecoding"
    };
    print_array(int_array, 8,
                sizeof(int), int print);
    print_array(str_array, 8,
        sizeof(str_array[0]), str_print);
    return 0;
}
```

```
2 50 3 -7 2 9 0 0
zzz
zzzz
xxxxxxxxx
p
abc
ccckkk
aaa
mmwordilovecoding

请按任意键继续. . .
```

例1-两个未知类型元素的交换

```
#include <stdio.h>
void swap(char *a, char *b, int width){
    char tmp;
    while(width){
        tmp = *a;
        *a = *b;
        *b = tmp;
        a++;
        b++;
        width--;
    }
}
int main(){
    int a = 5;
    int b = 10;
    char str1[10] = "abcde";
    char str2[10] = "xxx";
    printf("a = %d b = %d\n", a, b);
    printf("str1 = %s str2 = %s\n", str1, str2);
    swap((char *)&a, (char *)&b, sizeof(int));
    swap(str1, str2, sizeof(str1));
    printf("a = %d b = %d\n", a, b);
    printf("str1 = %s str2 = %s\n", str1, str2);
    return 0;
}
```

//对于两个元素的交换，就是对于这两个元素的占用的内存中每个字节的交换

```
a = 5 b = 10
str1 = abcde str2 = xxx
a = 10 b = 5
str1 = xxx str2 = abcde
请按任意键继续. . .
```

//用swap交换两个整型和两个字符串

例1-课堂练习

```
void sort_array(void *base, int num, int width,
                int (*compare)(const void *, const void *)){
    int i, j;
    void *temp = 1
    for (i = 0; i < num; i++){
        for (j = i + 1; j < num; j++){
            void *element1 = base + i * width;
            void *element2 = 2
            if ( 3 > 0){
                swap(temp, element1, width);
                swap(element1, element2, width);
                swap(element2, temp, width);
            }
        }
    }
    free(temp);
}
```

```
//对整数排序的回调函数
int int_cmp(const void *a, const void *b){
    return *(int *)a - *(int *)b;
}
//对浮点数排序的回调函数
int double_cmp(const void *a, const void *b){
    return 4
}
//对字符串数组排序的回调函数
int str_cmp(const void *a, const void *b){
    return strcmp((char *)a, (char *)b);
}
//对字符串指针数组排序的回调函数
int str_ptr_cmp(const void *a, const void *b){
    return 5
}
```

3分钟，填写代码，有问题提出！

例1-实现

```
void sort_array(void *base, int num, int width,
                int (*compare)(const void *, const void *)){
    int i, j;
    void *temp = malloc(width);

    for (i = 0; i < num; i++){
        for (j = i + 1; j < num; j++){
            void *element1 = base + i * width;

            void *element2 = base + j * width;

            if ( compare(element1, element2) > 0){
                swap(temp, element1, width);
                swap(element1, element2, width);
                swap(element2, temp, width);
            }
        }
    }
    free(temp);
}
```

```
//对整数排序的回调函数
int int_cmp(const void *a, const void *b){
    return *(int *)a - *(int *)b;
}
//对浮点数排序的回调函数
int double_cmp(const void *a, const void *b){
    return *(double *)a > *(double *)b ? 1 : -1;
}
//对字符串数组排序的回调函数
int str_cmp(const void *a, const void *b){
    return strcmp((char *)a, (char *)b);
}
//对字符串指针数组排序的回调函数
int str_ptr_cmp(const void *a, const void *b){
    return strcmp(*(char **)a, *(char **)b);
}
```

函数中的静态变量

在函数体内中声明的变量是**自动变量**，即在声明时**自动创建**，在函数退出前**自动销毁**。在某些情况下，希望再退出一个函数调用后，存储在某些变量中的数据**仍然保存**在内存中，下次调用这个函数时**仍能使用**这份数据。

这时就要把变量声明为**静态变量**。**static关键字**可将变量指定为静态变量，若在函数的作用域内定义静态变量，当函数退出后，静态变量**不会销毁**。声明在函数中的静态变量只在该函数**第一次执行**时初始化一次，虽然它只在包含它的函数中可见，但它是一个**全局变量**。

```c
#include <stdio.h>

void func_test1(){
    int count = 0;
    count++;
    printf("func_test1 count = %d\n", count);
}

void func_test2(){
    static int count = 0;
    count++;
    printf("func_test2 count = %d\n", count);
}
```

//自动变量count，
函数每次创建时都创建
函数每次结束前都销毁

//静态变量count，只在函数第一次
调用时创建并初始化函数结束后不销毁

```c
int main(){
    int i;
    for (i = 0; i < 5; i++){
        func_test1();
    }
    printf("\n");
    for (i = 0; i < 5; i++){
        func_test2();
    }
    return 0;
}
```

```
func_test1 count = 1
func_test1 count = 1
func_test1 count = 1
func_test1 count = 1
func_test1 count = 1

func_test2 count = 1
func_test2 count = 2
func_test2 count = 3
func_test2 count = 4
func_test2 count = 5
请按任意键继续. . .
```

函数间共享变量:全局变量

在某些情况下，我们希望在所有的函数之间**共享变量**，这种变量称为**全局变量**，声明为全局的变量可以在它**声明后**的**任意位置**访问。全局变量的声明方式与一般变量相同，它**声明的位置**决定了该变量是否是全局的或可以被哪些函数所访问。

一般来讲，我们**不建议**将变量声明为**全局**，因为这样的变量是**线程不安全**的，即**多线程调用**时可能出现**不一致**的结果，这种问题在测试中常被称为**单多线程结果不一致**。若将变量声明为全局且希望保证结果的一致性，则需要加**线程锁**，而**过多的使用锁**又会导致程序性能问题。

```c
#include <stdio.h>

int count = 0;

void func_test1(){
    count++;
    printf("func_test1 count = %d\n", count);
}

void func_test2(){
    count++;
    printf("func_test2 count = %d\n", count);
}
```

//全局变量count，可被所有在它之后声明的函数所使用(或利用extern进行包含)

//访问全局的count

//访问全局的count

```c
int main(){
    int i;
    for (i = 0; i < 5; i++){
        func_test1();
    }
    printf("\n");
    for (i = 0; i < 5; i++){
        func_test2();
    }
    return 0;
}
```

```
func_test1 count = 1
func_test1 count = 2
func_test1 count = 3
func_test1 count = 4
func_test1 count = 5

func_test2 count = 6
func_test2 count = 7
func_test2 count = 8
func_test2 count = 9
func_test2 count = 10
请按任意键继续. . .
```

main函数

main函数程序**执行的起点**，main函数也可以有**参数列表**，即在**命令执行时**(程序执行时)直接给程序**传递参数**。
main函数的**参数**:int main(int argc, char *argv[])
main函数一般**没有参数**或有**两个参数**(虽然可以定义**更多的参数**或参数**设置的顺序**不一样也可以编译通过，但是这么就做使得参数就没什么用处了)。main函数默认的参数是**argc**与**argv**(可以不叫这两个名字)，代表命令提示符下输入的**数量**与具体的**字符串**默认**第1个字符串**是该**程序的名字**，如果希望给程序输入更多的字符串，则需要在**cmd命令提示符**下输入(Windows)。
main函数的**返回值**:
main函数的返回值返回给**操作系统**，标准的main函数应该**返回int**，当然也可以返回double、char*或者什么都不返回void，但这么做操作系统就**无法理解**该程序的返回值了(实际上这么做也没有什么用处)。main函数的返回值会在**程序运行结束**后，返回给操作系统，在windows下使用环境变量**%errorlevel%**获取。

```c
#include <stdio.h>
//从控制台输入参数的个数
int main(int argc, char *argv[]){
    printf("argc = %d\n", argc);
    int i;
    for (i = 0; i < argc; i++){
        printf("%s\n", argv[i]);
    }
    return 999;
}
```

//参数的字符串指针

```
argc = 1
C:\C语言程序设计\main函数.exe
请按任意键继续. . .
```

```
C:\C语言程序设计>main函数.exe aaa b c d eeeee
argc = 6
main函数.exe
aaa
b
c
d
eeeee

C:\C语言程序设计>echo %errorlevel%
999

C:\C语言程序设计>
```

参数数量可变的函数

在开发程序时，有时设计函数时需要函数的**参数数量与类型**是可变的，例如输入函数**scanf**与输出函数**printf**。标准库**<stdarg.h>**中提供了相应的方法来实现参数数量可变的函数。在设计时函数原型时，固定参数需要几个写几个，**至少需要**提供1个固定参数，**"..."**表示**可变参数**。

参数数量可变的**函数原型**如下:

函数返回值 函数名(参数1类型 参数1名称, 参数2类型 参数2名称, ...);

在获取**参数列表**时，需要创建**va_list类型**的变量，并同时使用**三个宏**(注意不是函数):

va_list parg;定义**参数列表parg**。

va_start(参数列表parg, 最后一个固定的函数名):这个宏接收两个参数，分别是**参数列表指针**和该函数原型中**最后一个固定参数名**。

va_arg(参数列表parg，类型):

这个宏接收两个参数，分别是**参数列表指针**与当前获取的这个**参数的类型**。实际上，虽然每次输入的**参数数量与类型**可以**不同**，但程序**必须确定**具体需要**读入多少个参数**与**每个参数的类型**。

va_end():

结束时需要调用va_end()宏，它会重置parg为**NULL**，函数结束前该宏必须调用，否则程序后面可能无法正常工作。

```c
#include <stdio.h>
#include <stdarg.h>  //固定参数，至少有1个
double average( int value , ... ){
    double sum = value;
    int count = 1;
    va_list parg;  //参数列表指针
    va_start(parg, value); //最后一个固定参数
    value = va_arg(parg, int); //通过参数列表指针，获取下一个参数
    while (value != -1){
        sum += value; //只要获取到的参数值不是-1，就继续 计算并获取新的参数
        count++;
        value = va_arg(parg, int);
    }
    va_end(parg);  //函数结束前，要写上这句话
    return sum / count;
}
int main(){
    printf("%lf\n", average(1, -1));
    printf("%lf\n", average(1, 2, -1));
    printf("%lf\n", average(1, 2, 3, -1));
    printf("%lf\n", average(1, 2, 3, 4, -1));
    printf("%lf\n", average(1, 2, 3, 4, 5, -1));
    return 0;
}
```

```
1.000000
1.500000
2.000000
2.500000
3.000000
请按任意键继续. . .
```

例2-简单版本的printf函数

设计一个**简单版本**的printf函数:simple_print()，simple_print使用方法与printf完全一样，但只支持打印**整型、字符型、字符串**三种数据类型的变量打印，对应的**转换说明符**为%d、%c、%s。在**打印过程中**只可以使用int putchar(int ch);函数，它的作用是在屏幕上以**字符形式**打印ch。在simple_print中，**%号后**的字符只需要考虑**d、c、s**三个字符。程序不可包含<stdio.h>头文件，putchar的函数原型在程序中给出。

```
#include <stdarg.h>
int putchar(int ch);
void simple_print(const char *control, ...){
}

int main(){
    char name[20] = "LinMu";
    int age = 28;
    simple_print(
     "My name is %s, I'm %d years old.\n", name, age);
    simple_print("My name is spelled:\n");
    int i = 0;
    while(name[i]){
        simple_print("%c\n", name[i]);
        i++;
    }
    return 0;
}
```

//打印只可使用在屏幕上

打印一个字符putchar函数

```
My name is LinMu, I'm 28 years old.
My name is spelled:
L
i
n
M
u
请按任意键继续. . .
```

例2-思考与分析

```
//该字符串指向需要打印的字符串，其中转换说明符
支持%d、%c、%s
void simple_print( const char *control , ... ){
}
//需要打印的变量，变量的个 数、类型均不确定，由control
中的转换说明符来说明
```

```
simple_print("My name is %s, I'm %d years old.\n", name, age);
```

```
My name is LinMu, I'm 28 years old.
```

思考：

1.在打印过程中，如何将control字符串中的**常量字符**、并根据**转换说明符**将参数列表的**其他变量**打印出来?

2.如何利用putchar函数将**整数变量**或**字符串变量**打印在屏幕上。

3.**整体算法**是怎样的?

例2-打印整数与字符串

```
int putchar(int ch);
void print_int(int value){
    char temp[30] = {0};
    int cnt = 0;
    while(value){
        temp[cnt++] = 1
        value = 2
    }
    while(cnt){
        cnt--;
        putchar( 3 );
    }
}
void print_string(const char *value){
    while(*value){
        putchar(*value);
        value++;
    }
}
```

```
int main(){
    int a = 1234567890;
    char *str = "abcdefg ### ppp";
    print_int(a);
    putchar('\n');
    print_string(str);
    putchar('\n');
    return 0;
}
```

```
1234567890
abcdefg ### ppp
请按任意键继续. . .
```

3分钟，填写代码，有问题提出！

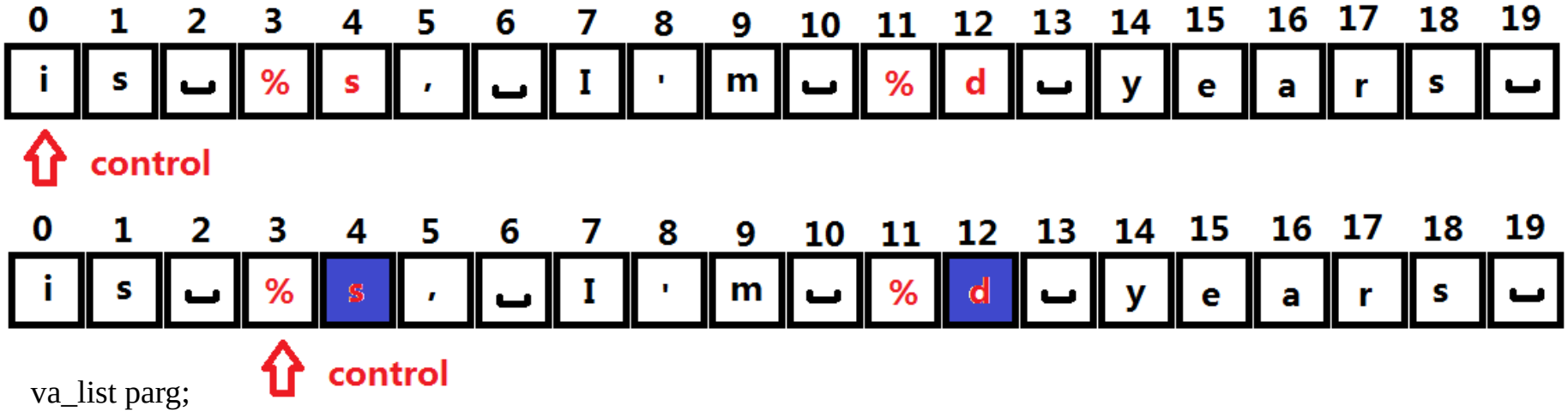
例2-打印整数与字符串，实现

```
int putchar(int ch);
void print_int(int value){
    char temp[30] = {0};
    int cnt = 0;
    while(value){
        temp[cnt++] = value % 10;
        value = value / 10;
    }
    while(cnt){
        cnt--;
        putchar( temp[cnt] + '0' );
    }
}
void print_string(const char *value){
    while(*value){
        putchar(*value);
        value++;
    }
}
```

```
int main(){
    int a = 1234567890;
    char *str = "abcdefg ### ppp";
    print_int(a);
    putchar('\n');
    print_string(str);
    putchar('\n');
    return 0;
}
```

```
1234567890
abcdefg ### ppp
请按任意键继续. . .
```

例2-算法设计



va_list parg;
va_start(parg, control);
使用control指针**遍历**字符串:
当control指向的字符是**%号**时，将**%号后面的字符**取出:

设置char type为control指针的**后一个字符**，
如果type是'd'，**调用va_arg**获取参数，存储在int value中，调用print_int打印value；
如果type是'c'，**调用va_arg**获取参数，存储在int value中，调用putchar打印value；
如果type是's'，**调用va_arg**获取参数，存储在const char * value中，调用print_string打印value；
当type是'd'、'c'、's'时，指针**向后移动**；

否则，直接通过**putchar**将该字符输出。
control指针**向后移动**。
va_end(parg);

例2-课堂练习

```
void simple_print(const char *control, ...){
    va_list parg;
    va_start(parg, 1 );
    while(*control){
        if (*control == '%'){
            char type = 2
            if (type == 'd'){
                int value = va_arg(parg, int);
                print_int(value);
            }
            else if (type == 'c'){
                int value = va_arg(parg, int);
                putchar(value);
            }
            else if (type == 's'){
                const char *value = va_arg(parg, 3 );
                print_string(value);
            }
            if (type == 'd' || type == 'c' || type == 's'){
                4
            }
        }
        else{
            5
        }
        control++;
    }
    va_end(parg);
}
```

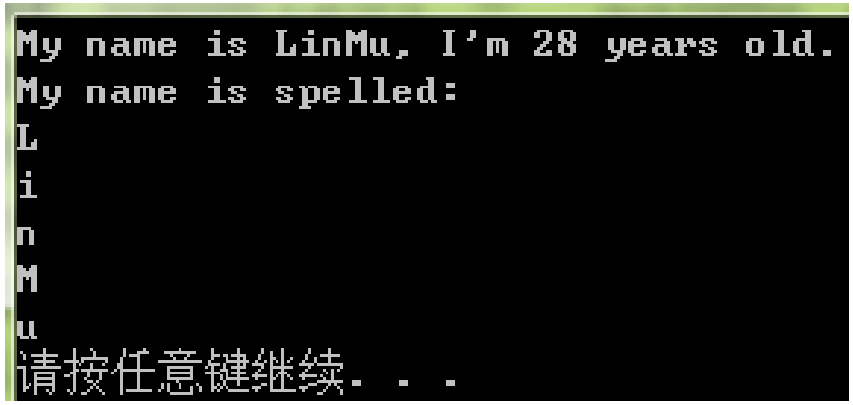
3分钟，填写代码，有问题提出！

例2-实现

```
void simple_print(const char *control, ...){
    va_list parg;
    va_start(parg, control);
    while(*control){
        if (*control == '%'){
            char type = *(control+1);
            if (type == 'd'){
                int value = va_arg(parg, int);
                print_int(value);
            }
            else if (type == 'c'){
                int value = va_arg(parg, int);
                putchar(value);
            }
            else if (type == 's'){
                const char *value = va_arg(parg, const char *);
                print_string(value);
            }
            if (type == 'd' || type == 'c' || type == 's'){
                control++;
            }
        }
        else{
            putchar(*control);
        }
        control++;
    }
    va_end(parg);
}
```

例2-测试

```
int main(){
    char name[20] = "LinMu";
    int age = 28;
    simple_print("My name is %s, I'm %d years old.\n", name, age);
    simple_print("My name is spelled:\n");
    int i = 0;
    while(name[i]){
        simple_print("%c\n", name[i]);
        i++;
    }
    return 0;
}
```



课间休息10分钟！

有问题提出！

C语言库函数:快速排序qsort

C语言函数库自带的**快速排序**函数，**时间复杂度nlogn**，函数原型与使用方法与例1的sort_array**完全一样**，使用qsort时，需要包含头文件**<stdlib.h>**。**函数原型:**

void qsort(void *base, int num, int width, int (*compare)(const void *, const void *))

void *base : 指向待排序的数组；int num: 待排序数组的元素个数；int width: 待排序数组的元素大小。

int (*compare)(const void *, const void *): 排序时的回调函数，用户传进来一个回调函数，在比较两元素的时候，使用这个回调函数对元素进行比较。

```c
#include <stdio.h>
#include <stdlib.h>

int int_cmp(const void *a, const void *b){
    return *(int *)a - *(int *)b;
}

int double_cmp(const void *a, const void *b){
    return *(double *)a > *(double *)b ? 1 : -1;
}

int main(){
    int int_array[8] = {2, 50, 3, -7, 2, 9, 0, 0};
    double double_array[8] = {2.5, 13.12, -0.7, 0, -50, 9, -0.123, 0};
    qsort(int_array, 8, sizeof(int), int_cmp);
    printf("int_array:\n");
    for (i = 0; i < 8; i++){
        printf("%d ", int_array[i]);
    }
    printf("\n\n");
    qsort(double_array, 8, sizeof(double), double_cmp);
    printf("double_array:\n");
    for (i = 0; i < 8; i++){
        printf("%lf ", double_array[i]);
    }
    printf("\n\n");
    return 0;
}
```

```
int_array:
-7 0 0 2 2 3 9 50

double_array:
-50.000000 -0.700000 -0.123000 0.000000 0.000000 2.500000 9.000000 13.120000

请按任意键继续. . .
```

C语言库函数:二分查找bsearch

C语言函数库自带的**二分查找**函数，**时间复杂度logn**，函数原型与使用方法与使用qsort类似，只是多了一个**参数key**，它指向**待查找元素**。如果在base数组中**找到key**，返回对应的**数据地址**，否则返回空。一般bsearch与qsort结合起来使用，即**先快排后二分**。

函数原型:void *bsearch(const void *key, const void *base, int num, int width, int (*compare)(const void *, const void *));

```c
#include <stdio.h>
#include <stdlib.h>

int int_cmp(const void *a, const void *b){
    return *(int *)a - *(int *)b;
}
int main(){
    int int_array[8] = {2, 50, 3, -7, 2, 9, 0, 0};
    int key[5] = {3, 7, 9, 0 , 5};
    qsort(int_array, 8, sizeof(int), int_cmp);   //快速排序
    int i;
    printf("array:\n");
    for (i = 0; i < 8; i++){
        printf("%d %p\n", int_array[i], &int_array[i]);
    }
    printf("\n");

    printf("result:\n");
    for (i = 0; i < 5; i++){                     //二分查找
        void *res = bsearch(&key[i], int_array, 8, sizeof(int), int_cmp);

        if (res){
            printf("%d %p\n", *(int *)res, res);
        }

    }
    return 0;
}
```

```
array:
-7 0028FF10
0 0028FF14
0 0028FF18
2 0028FF1C
2 0028FF20
3 0028FF24
9 0028FF28
50 0028FF2C

result:
3 0028FF24
9 0028FF28
0 0028FF14
请按任意键继续. . .
```

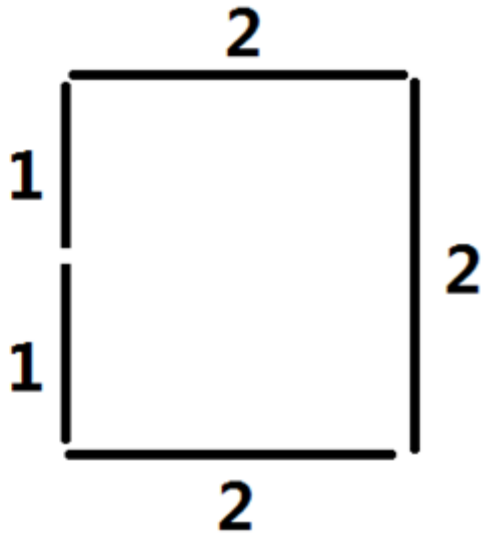
例3:火柴棍摆正方形

已知一个数组，保存了n个(n<=15)**火柴棍**，问可否使用这n个火柴棍摆成1个**正方形**？

[1, 1, 2, 2, 2] : true

[3, 3, 4, 4, 4] : false

[1, 1, 2, 4, 3, 2, 3] : true



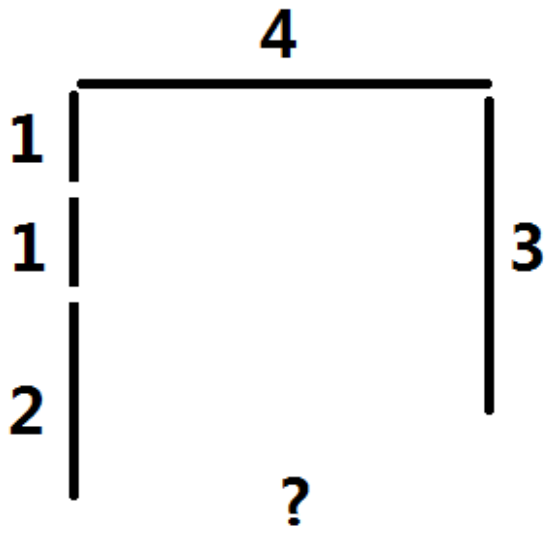

```
bool makesquare(int* nums, int numsSize) {

}
```

选自 **LeetCode 473. Matchsticks to Square**
https://leetcode.com/problems/matchsticks-to-square/description/
难度:**Medium**

例3:思考

n个火柴杆(n<=15)，每个**火柴杆**可以属于正方形的4个边中的其中1个。故，**暴力搜索**(回溯搜索)有4^15种可能。

[1, 1, 2, 4, 3, 2, 3]



思考:

1.回溯算法如何设计，与第5课，求子集题目有哪些**相似**的地方？

2.如何设计递归函数，递归的回溯搜索何时返回**真**，何时返回**假**？

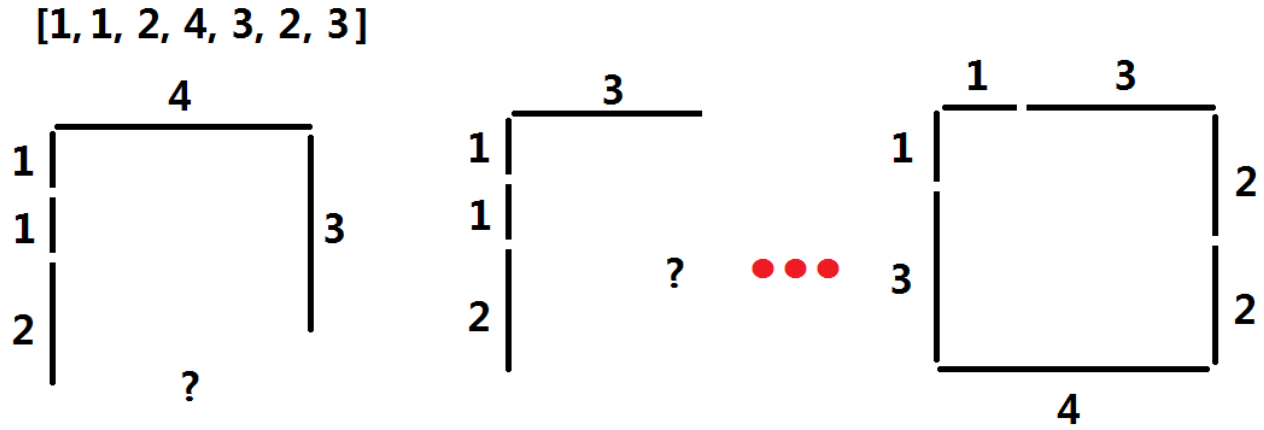
3.普通的回溯搜索可否解决该问题，如何对深度搜索如何优化**(剪枝)**，即可使得回溯搜索更加**高效**？

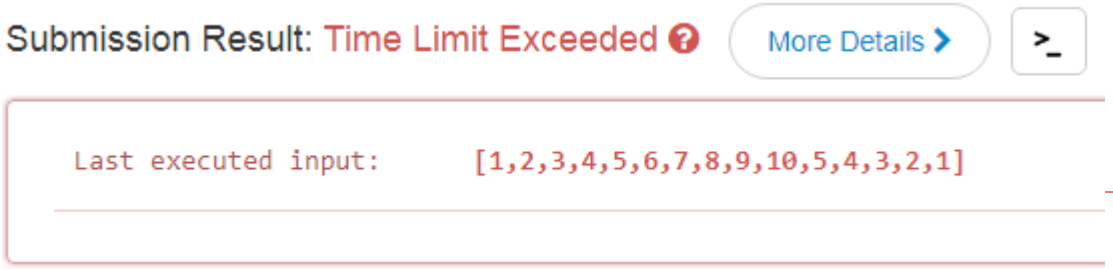
4.该题是否也可以用**位运算**的方式解决？

例3:无优化的回溯搜索

算法如下:

想象正方形的4条边即为4个桶，将每个火柴杆**回溯的放置**在每个桶中，在放完n个火柴杆后，检查4个桶中的火柴杆**长度和**是否相同，**相同**返回**真**，**否则**返回**假**；在回溯过程中，如果当前**所有可能**向后的**回溯**，都无法满足条件，即递归函数**最终**返回假。



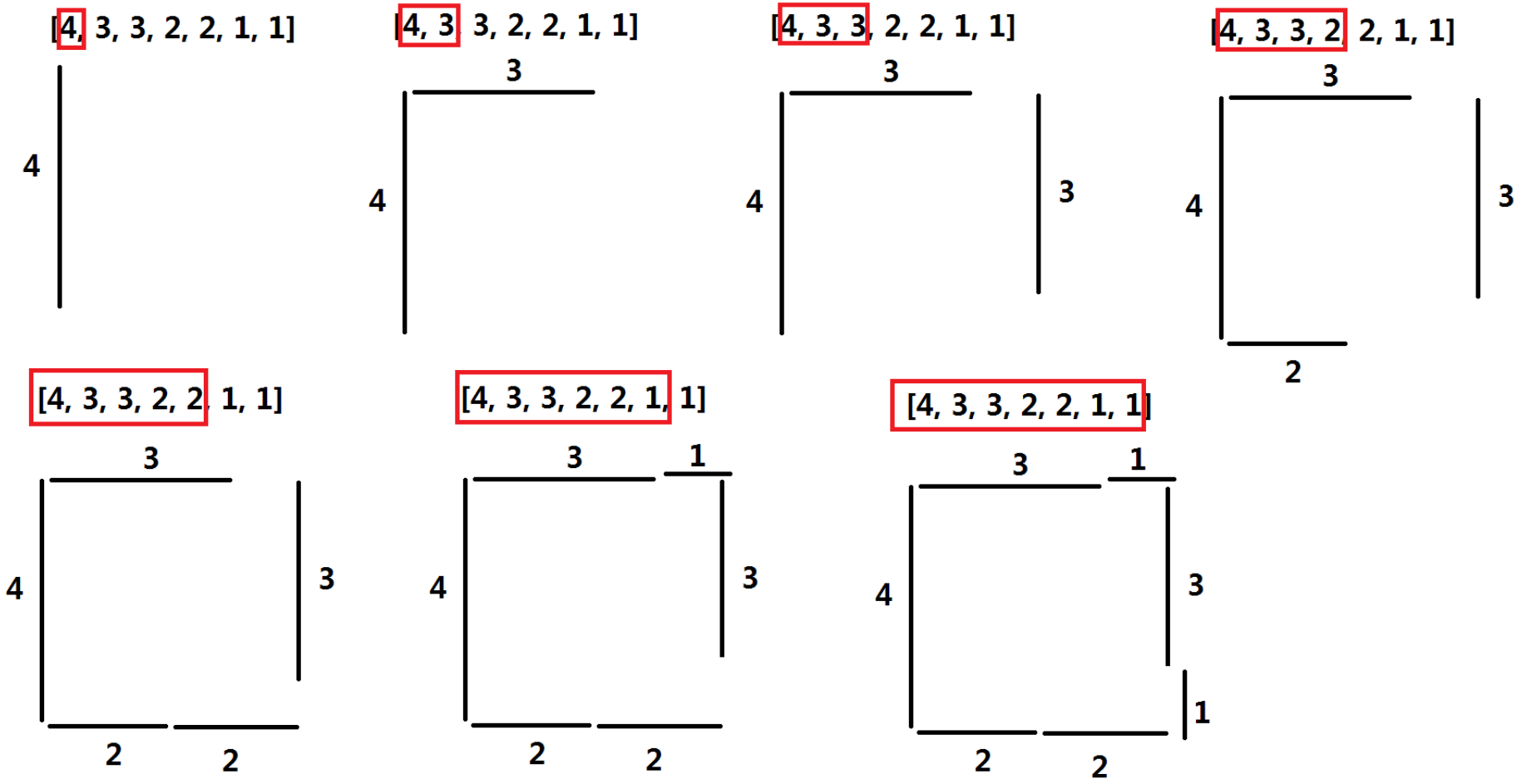


例3:优化与剪枝

优化1:n个火柴杆的总和**对4取余**需要为0，否则返回**假**。

优化2:火柴杆按照**从大到小**的顺序排序，先尝试大的减少**回溯**可能。

优化3:每次放置时，每条边上不可放置**超过总和**的1/4长度的火柴杆。



例3:优化与剪枝

```
#include <stdlib.h>
int int_cmp(const void *a, const void *b){
    return *(int *)b - *(int *)a;
}
bool makesquare(int* nums, int numsSize) {
    if (numsSize < 4){
        return false;    //数量不对，return
    }
    int sum = 0;
    int i;
    for (i = 0; i < numsSize; i++){
        sum += nums[i];
    }
    if (sum % 4){
        return false;    //无法被4整除，即无法摆到4条边上
    }                    //对火柴杆从大到小快速排序
    qsort(nums, numsSize, sizeof(int), int_cmp);
    int bucket[4] = {0};
    return generate(0, nums, numsSize, sum / 4, bucket);
}
```

例3:课堂练习

```
bool generate(int i, int nums[], int numsSize, int target, int bucket[]){
    if (          1          ){
        return bucket[0] == target && bucket[1] == target
            && bucket[2] == target && bucket[3] == target;
    }
    int j;
    for (j = 0; j < 4; j++){
        if (                2                ){
            continue;
        }
        bucket[j] += nums[i];
        if (generate(i + 1, nums, numsSize, target, bucket)){
            return true;
        }
        bucket[j] -= nums[i];
    }
    return false;
}
```

3分钟填写代码，
有问题随时提出！

例3:实现

```
bool generate(int i, int nums[], int numsSize, int target, int bucket[]){
    if ( i >= numsSize ){
        return bucket[0] == target && bucket[1] == target
            && bucket[2] == target && bucket[3] == target;
    }
    int j;
    for (j = 0; j < 4; j++){
        if ( bucket[j] + nums[i] > target ){
            continue;
        }
        bucket[j] += nums[i];
        if (generate(i + 1, nums, numsSize, target, bucket)){
            return true;
        }
        bucket[j] -= nums[i];
    }
    return false;
}
```

例3:位运算法

1.使用**位运算法**，构造出所有和为target(总和/ 4)的**子集**，存储在向量ok_subset中，这些是**候选的边**组合。

2.遍历所有的ok_subset，**两两进行对比**，如果ok_subset[i]和ok_subset[j]进行**与运算**的结果为0，则说明ok_subset[i]和ok_subset[j]表示的是**无交集**的两个集合(没有选择同样的火柴棍)，这两个集合可以代表两个可以**同时存在**的满足条件的边；将ok_subset[i]与ok_subset[j]**求或**，结果存储在ok_half中，它代表所有**满足一半结果**的情况。

3.遍历所有的ok_half，**两两进行对比**，如果ok_half[i]和ok_half[j]进行与运算的结果为0，则返回true(说明有4个满足条件的边，即可组成**正方形**)；否则返回false。

[1, 1, 2, 4, 3, 2, 3]

ok_subset:

[1, 1, 2, 4, 3, 2, 3]，集合代表值:7

[1, 1, 2, 4, 3, 2, 3]，集合代表值:8

[1, 1, 2, 4, 3, 2, 3]，集合代表值:17

[1, 1, 2, 4, 3, 2, 3]，集合代表值:18

[1, 1, 2, 4, 3, 2, 3]，集合代表值:35

[1, 1, 2, 4, 3, 2, 3]，集合代表值:65

[1, 1, 2, 4, 3, 2, 3]，集合代表值:66

ok_half :

[1, 1, 2, 4, 3, 2, 3]，集合代表值:15

[1, 1, 2, 4, 3, 2, 3]，集合代表值:25

…

[1, 1, 2, 4, 3, 2, 3]，集合代表值:102

找到了满足条件的！

例3:课堂练习

```
#define MAXN 500000

bool makesquare(int* nums, int numsSize) {
    if (numsSize < 4){
        return false;
    }
    int sum = 0;
    int i, j;
    for (i = 0; i < numsSize; i++){
        sum += nums[i];
    }
    if (sum % 4){
        return false;
    }
    int target = sum / 4;
    int ok_subset[MAXN] = {0};
    int ok_subset_len = 0;
    int ok_half[MAXN] = {0};
    int ok_half_len = 0;
    int all = [ 1 ]

    for (i = 0; i < all; i++){
        int sum = 0;
        for (j = 0; j < numsSize; j++) {
            if (i & (1 << j)){
                [ 2 ]
            }
        }
        if ( [ 3 ] ){
            ok_subset[ok_subset_len++] = i;
        }
    }
}
```

```
for (i = 0; i < ok_subset_len; i++){
    for (j = i + 1; j < ok_subset_len; j++){
        if ((ok_subset[i] & ok_subset[j]) == 0){
            ok_half[ok_half_len++] = [ 4 ]
        }
    }
}
for (i = 0; i < ok_half_len; i++){
    for (j = i + 1; j < ok_half_len; j++){
        if (( [ 5 ] ) == 0){
            return true;
        }
    }
}
return false;
```

5分钟填写代码，
有问题随时提出！

例3:实现

```
#define MAXN 500000

bool makesquare(int* nums, int numsSize) {
    if (numsSize < 4){
        return false;
    }
    int sum = 0;
    int i, j;
    for (i = 0; i < numsSize; i++){
        sum += nums[i];
    }
    if (sum % 4){
        return false;
    }
    int target = sum / 4;
    int ok_subset[MAXN] = {0};
    int ok_subset_len = 0;
    int ok_half[MAXN] = {0};
    int ok_half_len = 0;
    int all = 1 << numsSize;
    for (i = 0; i < all; i++){
        int sum = 0;
        for (j = 0; j < numsSize; j++) {
            if (i & (1 << j)){
                sum += nums[j];
            }
        }
        if (sum == target){
            ok_subset[ok_subset_len++] = i;
        }
    }
    for (i = 0; i < ok_subset_len; i++){
        for (j = i + 1; j < ok_subset_len; j++){
            if ((ok_subset[i] & ok_subset[j]) == 0){
                ok_half[ok_half_len++] = ok_subset[i] | ok_subset[j];
            }
        }
    }
    for (i = 0; i < ok_half_len; i++){
        for (j = i + 1; j < ok_half_len; j++){
            if ((ok_half[i] & ok_half[j]) == 0){
                return true;
            }
        }
    }
    return false;
}
```

[1, 1, 2, 4, 3, 2, 3]

ok_subset:

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:7

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:8

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:17

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:18

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:35

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:65

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:66

ok_half :

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:15

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:25

...

[1, 1, 2, 4, 3, 2, 3] , 集合代表值:102

找到了满足条件的！

例3:测试与leetcode提交结果

```
int main(){
    int nums[] = {1, 1, 2, 4, 3, 2, 3};
    printf("%d\n", makesquare(nums, 7));
    return 0;
}
```

```
1
请按任意键继续. . .
```

Matchsticks to Square

Submission Details

174 / 174 test cases passed. Status: Accepted

Runtime: **49 ms** Submitted: **0 minutes ago**

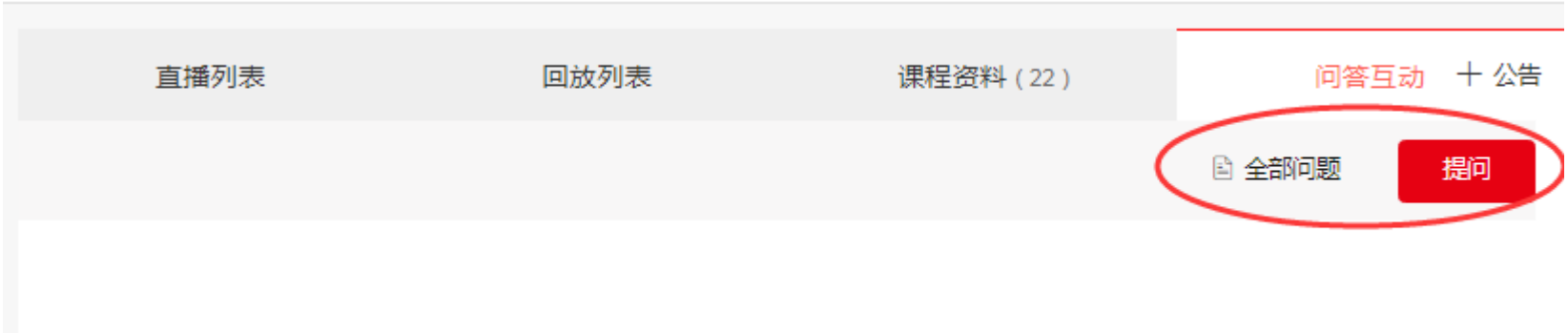
结束

非常感谢大家！

林沐

问答互动

在所报课的课程页面，
1、点击“全部问题”显示本课程所有学员提问的问题。
2、点击“提问”即可向该课程的老师和助教提问问题。



联系我们

小象学院：互联网新技术在线教育领航者

– 微信公众号：**小象学院**