实验名称：Bomblab 二进制炸弹实验

实验目的：

通过拆解给定的二进制炸弹程序，熟悉 Linux 系统的使用，掌握程序反汇编和逆向工程的基本方法，理解汇编语言，学习使用调试器的方法。

实验环境：

- 操作系统：Ubuntu 20.04.2
- GDB 版本：GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2

实验内容与步骤：

Bomb 预备工作：

首先使用 `mkdir ~/work/submit` 指令在 work 文件夹下创建 submit 文件夹，以用于后续答案和其他结课需求的提交。在输入该指令时，终端无后续提示，但可在左侧点击进入 work 文件夹，即可看到 submit 文件夹。

创建 submit 后，在终端使用 `tar -xvf ~/work/42024202.tar -C ~/work` 命令将 bomb 压缩包解压到 work 文件夹下，此时终端提示 bomb 和 bomb.c，并能在 work 目录下查找到该两文件。

使用 `objdump -d ~/work/bomb > ~/work/bomb.s` 将 bomb 进行反汇编，生成 bomb.s 文件，至此 bomb 已经生成，使用指令 `gdb bomb` 可正常拆弹。

Phase 0:

```
0804958d <phase_0>:
 804958d:    f3 0f 1e fb          endbr32
 8049591:    55                   push   %ebp
 8049592:    89 e5                mov    %esp,%ebp
 8049594:    83 ec 08             sub    $0x8,%esp
 8049597:    83 ec 08             sub    $0x8,%esp
 804959a:    68 d4 b1 04 08       push   $0x804b1d4
 804959f:    ff 75 08             pushl  0x8(%ebp)
 80495a2:    e8 bd 08 00 00       call   8049e64 <strings_not_equal>

 80495a7:    83 c4 10             add    $0x10,%esp
 80495aa:    85 c0                test   %eax,%eax
 80495ac:    74 0c                je     80495ba <phase_0+0x2d>
 80495ae:    e8 31 0b 00 00       call   804a0e4 <explode_bomb>
 80495b3:    b8 00 00 00 00       mov    $0x0,%eax
 80495b8:    eb 05                jmp    80495bf <phase_0+0x32>
 80495ba:    b8 01 00 00 00       mov    $0x1,%eax
 80495bf:    c9                   leave
 80495c0:    c3                   ret
```

Phase_0 代码行数较少，故可快速浏览全代码内容。

```
804959a:   68 d4 b1 04 08        push   $0x804b1d4
804959f:   ff 75 08              pushl  0x8(%ebp)
80495a2:   e8 bd 08 00 00        call   8049e64 <strings_not_equal>
```

发现在 804959a 和 804959f 两个位置栈帧压入两个数据，并在 80495a2 调用了函数<strings not equal>，根据函数名称猜测，该函数可能是判断之前 0x804b1d4 和 0x8(%ebp)内容是否相等，且 0x8(%ebp)可能是用户输入值。

```
80495aa:   85 c0                 test   %eax,%eax
80495ac:   74 0c                 je     80495ba <phase_0+0x2d>
80495ae:   e8 31 0b 00 00        call   804a0e4 <explode_bomb>
```

向后分析代码可发现 80495aa-80495ae 为判断点，当%eax=1 时，test 命令将会使得 ZF=0，从而在下步 je 命令跳转到 80495ba 从而避免爆炸点。

向上浏览，发现%eax 的值在上文被函数<strings_not_equal>写入，所以炸弹是否爆炸取决于该函数的返回值。

综合可得，该 phase 的目的在于让用户输入一串已预定的字符串，并将该字符串与内存中已经存好的答案字符串进行比较，若二者相同则解除炸弹，反之爆炸。

而通过浏览代码发现，内存中已经存好的答案已被存入到明码地址 0x804b1d4 中，通过查找该地址内容即可获得答案字符串。

```
(gdb) x/s 0x804b1d4
0x804b1d4:      "Why Dynamic Memory Allocation?"
```

发现答案字符串为“Why Dynamic Memory Allocation?”

运行 bomb，将该答案输入 phase_0

```
(gdb) r
Starting program: /home/jovyan/work/bomb
Welcome to my fiendish little bomb. You have 7 phases with
which to blow yourself up. Have a nice day!
Why Dynamic Memory Allocation?
Well done! You seem to have warmed up!
```

发现 phase_0 已被成功拆除，则该题答案即为

Why Dynamic Memory Allocation?

Phase_1：

首先根据之前拆弹文档中的要求，可以得知该题需要输入两个符合要求的浮点整数。

首先自上到下浏览代码，以 push 和 call 两个命令为重点。

```
80495e9:   8d 45 e0             lea    -0x20(%ebp),%eax
80495ec:   50                   push   %eax
80495ed:   8d 45 dc             lea    -0x24(%ebp),%eax
80495f0:   50                   push   %eax
```

此处发现两个被压入栈帧的参数，两个地址-0x20(%ebp)和-0x24%ebp)疑似为用户输入的两个数值的存放地址。

```
80495f1:   68 f3 b1 04 08       push   $0x804b1f3
80495f6:   ff 75 d4             pushl  -0x2c(%ebp)
80495f9:   e8 f2 fb ff ff       call   80491f0 <__isoc99_sscanf@plt>
```

发现第一个明码地址 0x804b1f3 和函数<__isoc99_sscanf@plt>，根据该

名称可以判断该函数应是用户输入数字个数，而通过 gdb 查看上一个明码地址可以得到“%d %d”，发现该题要求用户输入两个整数。

```
8049601:   83 f8 02              cmp    $0x2,%eax
8049604:   74 0c                 je     8049612 <phase_1+0x51>
8049606:   e8 d9 0a 00 00        call   804a0e4 <explode_bomb>
```

向下浏览代码，发现第一个 cmp 函数，判断 eax 和 2 是否相等，若相等则跳转炸弹，反之炸弹爆炸。不难发现 eax 的值来源于上一个函数的返回值，若用户输入两个值则返回 2 跳转炸弹，反之炸弹爆炸。

```
8049612:   8d 45 e8              lea    -0x18(%ebp),%eax
//eax=ebp-0x18
 8049615:  83 c0 04              add    $0x4,%eax
//eax+=0x4
 8049618:  8b 10                 mov    (%eax),%edx
//edx=eax
 804961a:  8b 45 dc              mov -0x24(%ebp),%eax
//eax=ebp-0x24=iuput 1
 804961d:  39 c2                 cmp    %eax,%edx
//compare eax==edx
 804961f:  75 0c                 jne    804962d <phase_1+0x6c>

 8049621:  8d 45 e8              lea    -0x18(%ebp),%eax
 8049624:  8b 10                 mov    (%eax),%edx
//edx=eax
 8049626:  8b 45 e0              mov -0x20(%ebp),%eax
//eax=ebp-0x20=input 2
 8049629:  39 c2                 cmp    %eax,%edx
 804962b:  74 0c                 je     8049639 <phase_1+0x78>
 804962d:  e8 b2 0a 00 00        call   804a0e4 <explode_bomb>
```

向下浏览代码，发现两个相似结构，即判断两个数字是否相等，若不相等则炸弹爆炸。

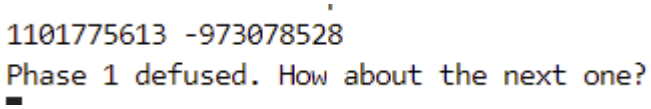
在上半段代码中，发现是比较用户输入的第一个数和（%edx）的值，则在 0x804961a 打断点并运行程序，直至程序停止在断点处，利用 layout regs 查看（%edx）的值

```
edx            0x41abc2fd       1101775613
ebp            0xffffd438       0xffffd438
```

发现在判断用户输入的第一个数和（%edx）的值之前，（%edx）存放的值为1101775613，说明用户输入的第一个数应为该值，才能使得炸弹被跳转。

同理判断代码段下半段，发现是比较用户输入的第二个数和（%edx）的值，再次打断点（此处打断点后运行时需要先输入已经知道的一个数以跳过第一个炸弹，才能顺利运行到第二处判断）并查询判断前（%edx）的值

不难得到用户输入的第二个数应为-973078528，综合可得答案为1101775613 -973078528，运行炸弹判断可以得到



Phase_2：

根据题目提示，该题需要输入满足某循环的一个数字序列。

向下浏览代码可以发现

```
804966f:   6a 09                push   $0x9
//n=9
8049671:   8d 45 d0             lea    -0x30(%ebp),%eax
//eax=ebp-0x30 数组首地址
8049674:   50                   push   %eax
8049675:   ff 75 c4             pushl  -0x3c(%ebp)
8049678:   e8 25 07 00 00       call   8049da2 <read_n_numbers>
```

代码向内压入了一个整数 9 和一个地址 ebp-0x30，并且调用了函数<read_n_numbers>，通过函数名可以猜测该函数为读入 n 个数字，而前文压入的 9 可能就是 n，但还需要进一步结合后文来看。

```
804968b:   8b 45 d0             mov    -0x30(%ebp),%eax
804968e:   83 f8 09             cmp    $0x9,%eax
8049691:   75 08                jne    804969b <phase_2+0x4a>
```

此处可以发现代码将刚才的地址-0x30(%ebp)赋给 eax，并将 eax 与 9 比较，若不相等则跳转到炸弹，说明-0x30(%ebp)内的数字需要等于 9 才能进行下一步操作，这里可以猜测猜测 9 为循环的第一个数。

```
8049693:   8b 45 d4              mov    -0x2c(%ebp),%eax
8049696:   83 f8 12              cmp    $0x12,%eax
8049699:   74 0c                 je     80496a7 <phase_2+0x56>
804969b:   e8 44 0a 00 00        call   804a0e4 <explode_bomb>
```

向后浏览可以看到 eax 重新被赋予了-0x2c(%ebp)的值，并且与 12 比较，若相等则跳过炸弹，说明-x2c(%ebp)的值需要等于 12 才能进行下一步操作。

```
80496e1:   83 45 cc 01           addl   $0x1,-0x34(%ebp)
//i++
80496e5:   83 7d cc 08           cmpl $0x8,-0x34(%ebp)
//ebp-0x34=i<=8
80496e9:   7e c5                 jle    80496b0 <phase_2+0x5f>
```

跳过中间疑似一段计算的代码，再向下看，发现到计算后有-0x34(%ebp)自增，并且判断-0x34(%ebp)是否小于等于 8，若小于则跳转到上文循环中。

基本可以确定上文循环是一个类似 for 循环且等同于 for(int i=1;i<=8;i++)结构，并且循环判断输入的 9 位数是否符合要求，所以此处假设-0x34(%ebp)地址存放为 i，再返回上文查看循环。

```
80496a7:   c7 45 cc 02 00 00 00  movl   $0x2,-0x34(%ebp)
//ebp-0x34=2
80496ae:   eb 35                 jmp    80496e5 <phase_2+0x94>
```

回到之前查看第一个数为 9 且第二个数为 12 的代码行，发现在这之后有一个将 i 置为 2 的操作，说明在此之前的 9 和 12 分别为 a[0]和[1]，且-0x30(%ebp)的位置即为数组首地址。此处判断完前两个数是否相等后，则开辟一个 for(int i=2;i<=8;i++)的 for 循环用于判断后 7 位数是否符合要求。

```
80496b0:   8b 45 cc              mov    -0x34(%ebp),%eax
//eax=i
80496b3:   8b 44 85 d0           mov -0x30(%ebp,%eax,4),%eax
```

```
//eax=a[i]
80496b7:   8b 55 cc              mov    -0x34(%ebp),%edx
//edx=i
80496ba:   83 ea 02              sub    $0x2,%edx
//edx=i-2
 0496bd:   8b 54 95 d0           mov  -0x30(%ebp,%edx,4),%edx
//edx=ebp-0x30+4*(i-2)=a[i-2]
80496c1:   89 d1                 mov    %edx,%ecx
//ecx=edx=a[i-2]
80496c3:   d1 f9                 sar    %ecx
//ecx=ecx/2=a[i-2]/2
80496c5:   8b 55 cc              mov    -0x34(%ebp),%edx
//edx=i
80496c8:   83 ea 01              sub    $0x1,%edx
//edx=edx-1=i-1
80496cb:   8b 54 95 d0           mov  -0x30(%ebp,%edx,4),%edx
//edx=a[i-1]
80496cf:   01 ca                 add  %ecx,%edx
//edx+=ecx edx=a[i-1]+a[i-2]/2
80496d1:   39 d0                 cmp %edx,%eax
//check(edx==eax) a[i]=a[i-1]+a[i-2]/2

80496d3:   74 0c                 je     80496e1 <phase_2+0x90>
80496d5:   e8 0a 0a 00 00        call   804a0e4 <explode_bomb>
```

向后分析 for 循环的循环体可得，代码首先将 eax 赋予 i，并且在 eax 存放 a[i],而后将 edx 赋予 a[i-2]的值，再加入形参 ecx 存放 a[i-2]/2 的值，最后重新将 edx 赋予 a[i-1]的值，结合 eax、edx、ecx 和运算规律，可以具体得到该循环是一个 a[i]=a[i-1]+a[i-2]/2 的数学序列。

根据该规律写出 cpp 代码，并且运行得到前 9 位数，可以得到：

```cpp
#include<iostream>
using namespace std;
int main()
{
    int a[9]={0};
    a[0]=9;a[1]=18;

    for(int i=2;i<9;i++)
    {
        a[i]=a[i-1]+(a[i-2]/2);
    }

    for(int i=0;i<9;i++)
    {
        cout<<a[i]<<" ";
    }
    return 0;
}
```

由此可以得到前 9 位数：9 18 22 31 42 57 78 106 145

运行代入 phase_2 可以得到：

```
9 18 22 31 42 57 78 106 145
That's number 2.  Keep going!
```

Phase_3：

根据题目提示，发现此题要求输入满足 switch 条件的的字符串，而这个提示不足以告诉我们具体输入数字或是字符，但我们可以知道此 phase 主要是一个 switch 结构。

```
804971e:   8d 45 e8             lea    -0x18(%ebp),%eax
8049721:   50                   push   %eax
8049722:   8d 45 e4             lea    -0x1c(%ebp),%eax
8049725:   50                   push   %eax
```

首先从头开始浏览代码，不难发现代码在开始压入了两个值，分别存放在-0x18(%ebp)和-0x1c(%ebp)，这里可以猜测本炸弹同样是输入两个值。

```
8049726:   68 f3 b1 04 08       push   $0x804b1f3
//%d %d
804972b:   ff 75 d4             pushl  -0x2c(%ebp)
804972e:   e8 bd fa ff ff       call   80491f0 <__isoc99_sscanf@plt>
```

而后代码压入了一个明码，通过 gdb x/s 该明码可以得到“%d %d”，说明需要输入两个整数值，并且后续通过调用函数<__isoc99_sscanf@plt>来得到一个值并将这个值传入 eax。

```
8049736:   89 45 f0             mov    %eax,-0x10(%ebp)
8049739:   83 7d f0 01          cmpl   $0x1,-0x10(%ebp)
//check(ebp-0x10 > 1)输入 2 个数以上
```

```
804973d:   7f 0f                 jg     804974e <phase_3+0x4b>
804973f:   e8 a0 09 00 00        call   804a0e4 <explode_bomb>
```

这段代码将上文得到的 eax 与 1 相比，若大于 1 则跳转，否则炸弹爆炸，说明上文调用的函数用于得到输入值的个数，而输入的值为 2 个以上才可以正常进行下一步。

后而的代码是一串对于-0x14(%ebp)的运算，此处发现这个值对于目前代码行没有明显影响，故首先忽略不计，继续看后面的代码段。

```
80497ac:   8b 45 e4              mov    -0x1c(%ebp),%eax
//eax=ebp-0x1c
 80497af:  3d 9f 00 00 00        cmp    $0x9f,%eax
//check(eax <= 159)
 80497b4:  7f 16                 jg     80497cc <phase_3+0xc9>
//跳转炸弹
```

此处发现 eax 被赋予之前输入的第一个值，而 eax 需要与 0x9f 比较，大于 9f 则会跳转到炸弹点，所以需要输入第一个值小于等于 0x9f，通过进制转化可以得到第一个数应该小于等于 159。

```
80497c4:   8b 45 e8              mov    -0x18(%ebp),%eax
 80497c7:  39 45 ec              cmp    %eax,-0x14(%ebp)
//check(eax==a=2163)
 80497ca:  74 0c                 je     80497d8 <phase_3+0xd5>
 80497cc:  e8 13 09 00 00        call   804a0e4 <explode_bomb>
```

继续向下浏览代码可以看到，此处 eax 被赋予第二个输入值，而该输入值需要与-0x14(%ebp)比较，二者相等才可跳出炸弹点，所以我们返回上文寻找之前关于计算-0x14(%ebp)的代码串。

```
804974e:   c7 45 ec 00 00 00 00  movl   $0x0,-0x14(%ebp)
//x=0
804976c:   81 45 ec 49 03 00 00  addl   $0x349,-0x14(%ebp)
//x+0x349
8049773:   81 45 ec 95 02 00 00  addl   $0x295,-0x14(%ebp)
//x+0x349+0x295
804977a:   81 6d ec 49 03 00 00  subl   $0x349,-0x14(%ebp)
//x+0x295
8049781:   81 45 ec 49 03 00 00  addl   $0x349,-0x14(%ebp)
```

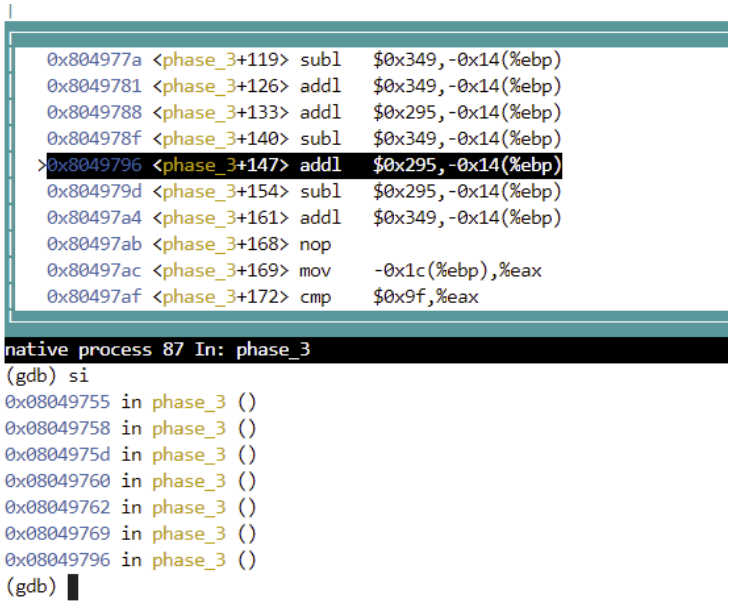
```
//x+0x349+0x295
8049788:   81 45 ec 95 02 00 00  addl   $0x295,-0x14(%ebp)
//x+0x295+0x349+0x349
804978f:   81 6d ec 49 03 00 00  subl   $0x349,-0x14(%ebp)
//x+0x295+0x295
8049796:   81 45 ec 95 02 00 00  addl   $0x295,-0x14(%ebp)
//x+0x295+0x295+0x295
804979d:   81 6d ec 95 02 00 00  subl   $0x295,-0x14(%ebp)
//x+0x295+0x295
80497a4:   81 45 ec 49 03 00 00  addl   $0x349,-0x14(%ebp)
//x+0x295+0x295+0x349
```

一通计算猛如虎，最后可以得到-0x14(%ebp)的值为 2163，所以此处可以得到用户输入的第二个值为 2163，代入 phase3 运行可以得到

```
159 2163

BOOM!!!
The bomb has blown up.
```

我们发现这道题并没有那么简单，重新 ctrl+f 查询有关于-0x14(%ebp)的运算语句可以得到，确实只有我们上述代码块中的代码内容，于是我们在进行这个代码块运算之前打入一个断点，一步步调试分析代码变化。

```
B+ 0x804974e <phase_3+75>  movl   $0x0,-0x14(%ebp)
   0x8049755 <phase_3+82>  mov    -0x1c(%ebp),%eax
   0x8049758 <phase_3+85>  sub    $0x99,%eax
   0x804975d <phase_3+90>  cmp    $0x8,%eax
   0x8049760 <phase_3+93>  ja     0x80497b8 <phase_3+181>
   0x8049762 <phase_3+95>  mov    0x804b1fc(,%eax,4),%eax
  >0x8049769 <phase_3+102> notrack jmp *%eax
   0x804976c <phase_3+105> addl   $0x349,-0x14(%ebp)
   0x8049773 <phase_3+112> addl   $0x295,-0x14(%ebp)
   0x804977a <phase_3+119> subl   $0x349,-0x14(%ebp)
native process 87 In: phase_3
(gdb) si
0x08049755 in phase_3 ()
0x08049758 in phase_3 ()
0x0804975d in phase_3 ()
0x08049760 in phase_3 ()
0x08049762 in phase_3 ()
0x08049769 in phase_3 ()
(gdb)
```



根据我们步步调试可得，在-0x14(%ebp)被置零后，代码由

```
8049769: 3e ff e0                   notrack jmp *%eax
```

直接从 8049769 跳转到了 8049796，这意味着之前的代码计算都不算数，计算-0x14(%ebp)需要直接从 8049796 开始重新计算，也就是下面这个代码块

```
8049796:   81 45 ec 95 02 00 00  addl   $0x295,-0x14(%ebp)
804979d:   81 6d ec 95 02 00 00  subl   $0x295,-0x14(%ebp)
80497a4:   81 45 ec 49 03 00 00  addl   $0x349,-0x14(%ebp)
```

对这三行代码重新计算可以得到，-0x14(%ebp)的值为 841，所以我们可以得到需要输入的两个整数为 159 841。

运行代入 phase_3 可以得到

```
159 841
Halfway there!
```

注意：此处第一个值只需要为小于等于 159 的值即可，但第二值需要等于 841，所以此题答案多解。

Phase_4：

根据题目提示，此题需要输入符合递归条件的字符串，但该字符串需要输入

数字或是字符尚不明确。

可以直接发现 phase_3 后紧接着是 func4，所以猜测这是 phase4 的递归函数，但暂不分析，先看函数主体

```
 80498a3:  8d 85 fc fe ff ff     lea    -0x104(%ebp),%eax
//第二个数
 80498a9:  50                    push   %eax
 80498aa:  8d 85 f8 fe ff ff     lea    -0x108(%ebp),%eax
//第一个数
 80498b0:  50                    push   %eax
 80498b1:  68 f3 b1 04 08        push   $0x804b1f3
//%d %d
 80498b6:  ff b5 f4 fe ff ff     pushl  -0x10c(%ebp)
 80498bc:  e8 2f f9 ff ff        call   80491f0 <__isoc99_sscanf@plt>

 80498c1:  83 c4 10              add    $0x10,%esp
 80498c4:  89 85 00 ff ff ff     mov    %eax,-0x100(%ebp)
 80498ca:  83 bd 00 ff ff ff 02  cmpl   $0x2,-0x100(%ebp)
 80498d1:  74 0f                 je     80498e2 <phase_4+0x7b>
 80498d3:  e8 0c 08 00 00        call   804a0e4 <explode_bomb>
```

分析方法和上一题相似，发现压入两个栈-0x104(%ebp)和-0x108(%ebp)应为两个输入值存放地址，并通过函数<__isoc99_sscanf@plt>取得输入个数值存入 eax，在后比较 eax 与 2 的值，若输入个数为 2 则跳出炸弹，若输入个数不等于 2 则炸弹爆炸，且通过 gdb 查看明码 0x804b1f3 得到应该输入两个整数。

```
80498e2:   8b 95 fc fe ff ff     mov    -0x104(%ebp),%edx
//edx=num2
80498e8:   8b 85 f8 fe ff ff     mov    -0x108(%ebp),%eax
//eax=num1
80498ee:   83 ec 04              sub    $0x4,%esp

80498f1:   52                    push   %edx
80498f2:   50                    push   %eax
80498f3:   8d 85 04 ff ff ff     lea    -0xfc(%ebp),%eax
80498f9:   50                    push   %eax
80498fa:   e8 f1 fe ff ff        call   80497f0 <func4>
```

此处将 edx、eax 分别赋值 num2、num1 压入栈中，并且重新压入一个确定

地址-0xfc(%ebp)，调用 func4，猜测 func4 是一个关于这三个栈的函数。

```
80498ff:   83 c4 10              add    $0x10,%esp
8049902:   3d d2 01 00 00        cmp    $0x1d2,%eax
//check(eax=466)
8049907:   74 0f                 je     8049918 <phase_4+0xb1>
```

```
804993d:   e8 ae fe ff ff        call   80497f0 <func4>
8049942:   83 c4 10              add    $0x10,%esp
8049945:   3d d2 01 00 00        cmp    $0x1d2,%eax
//check(eax!=466)
804994a:   75 0c                 jne    8049958 <phase_4+0xf1>
804994c:   e8 93 07 00 00        call   804a0e4 <explode_bomb>
```

```
804997e:   e8 6d fe ff ff        call   80497f0 <func4>
8049983:   83 c4 10              add    $0x10,%esp
8049986:   3d d2 01 00 00        cmp    $0x1d2,%eax
//check(eax!=466)
804998b:   75 0c                 jne    8049999 <phase_4+0x132>
804998d:   e8 52 07 00 00        call   804a0e4 <explode_bomb>
```

通过浏览 phase_4 全代码可以得到，phase 共调用了三次 func4 函数，并且比较了 3 次答案以判断是否能够跳过炸弹点，其中第一次为递归结果等于 466 可以跳过，其余两次为递归结果不等于 466 可以跳过。

```
8049958:   8b 85 fc fe ff ff     mov    -0x104(%ebp),%eax
//eax=num2
804995e:   83 f8 36              cmp    $0x36,%eax
//eax>54
8049961:   7f 36                 jg     8049999 <phase_4+0x132>
```

其中叠加一个判断条件，当输入的第二个值大于 54 时可以跳出递归。

所以该函数主体应该对应三个递归跳出条件，此时我们可以尝试分析 func4 的递归思路。

```
8049800:   01 d0                 add    %edx,%eax
//eax=eax+edx
8049802:   89 c2                 mov    %eax,%edx
//edx=eax
```

```
8049804:   c1 ea 1f              shr    $0x1f,%edx
//edx 右移 31 位 edx 置零
8049807:   01 d0                 add    %edx,%eax
//eax+=edx
8049809:   d1 f8                 sar    %eax
//eax/=2
804980b:   89 45 ec              mov    %eax,-0x14(%ebp)
//ebp-0x14=(num1+num2)/2
```

Func4 首先对两个参数进行了一通计算，最后的计算结果等于新设定一个形参-0x14(%ebp)用于存放(num1+num2)/2。

```
804980e:   8b 45 0c              mov    0xc(%ebp),%eax
//eax=num1
 8049811:  3b 45 10              cmp    0x10(%ebp),%eax
//num1<num2 jump
 8049814:  7c 13                 jl     8049829 <func4+0x39>
```

计算后来到第一个判定条件，此处将 eax 的值重新置为 num1，即判断 num1 是否小于 num2，即分开两个分支，分别是 num1<=num2 和 num1>num2 的情况。

此处我们先继续看 num1<=num2 的情况。

```
8049829:   83 ec 04              sub    $0x4,%esp
804982c:   ff 75 ec              pushl  -0x14(%ebp)
//(num1+num2)/2
804982f:   ff 75 0c              pushl  0xc(%ebp)
//num1
8049832:   ff 75 08              pushl  0x8(%ebp)
8049835:   e8 b6 ff ff ff        call   80497f0 <func4>
//func4(num1,(num1+num2)/2)
```

当 num1<=num2 时函数跳转到此处，不难发现代码重新将-0x14(%ebp)和 0xc(%ebp)压入栈中，并且还有 0x8(%ebp)压入，即此处应该调用 func4(num1,(num1+num2)/2)进行下一次递归，并且 0x8(%ebp)作为某一个特殊值参与 func4 计算。

```
804983a:   83 c4 10              add    $0x10,%esp
804983d:   89 45 f0              mov    %eax,-0x10(%ebp)
//ebp-0x10=func4(num1,(num+num2)/2)
```

```
8049840:   8b 45 ec             mov    -0x14(%ebp),%eax
//eax=(num1+num2)/2
8049843:   83 c0 01             add    $0x1,%eax
//eax+=1
8049846:   83 ec 04             sub    $0x4,%esp
8049849:   ff 75 10             pushl  0x10(%ebp)
804984c:   50                   push   %eax
804984d:   ff 75 08             pushl  0x8(%ebp)
8049850:   e8 9b ff ff ff       call   80497f0 <func4>
//func4(num1+num2)/2+1,num2)
```

在调用 func4 后该函数依然有后续步骤，即计算了(num1+num2)/2+1，并且将该值与 num2 和 0x8(%ebp) 一同压入栈中，调用 func4(num1+num2)/2+1,num2)继续计算。

```
8049855:   83 c4 10             add    $0x10,%esp
8049858:   89 45 f4             mov    %eax,-0xc(%ebp)
//储存到 ebp-0xc

804985b:   8b 45 f4             mov    -0xc(%ebp),%eax
//eax=ebp+0xc=func4((num1+num2)/2+1,num2)
804985e:   39 45 f0             cmp    %eax,-0x10(%ebp)
//func4((num+1num2)/2+1,num2) cmp func4(num1,(num1+num2)/2)
8049861:   0f 4d 45 f0          cmovge -0x10(%ebp),%eax
//若 ebp-0x10>=eax 则 eax=ebp-0x10
```

将上述两次函数调用值分别存入-0x10(%ebp)和 eax 中，比较 eax 与-0x10(%ebp) 的值，即比较 func4((num+1num2)/2+1,num2) 和 func4(num1,(num1+num2)/2)，利用 cmovge 命令 return 二者较大的一个值，并结束这次递归。

此时 num1<=num2 分析完毕，回到 num1>num2 的情况。

```
8049816:   8b 45 10             mov    0x10(%ebp),%eax
//eax=num2
8049819:   8d 14 85 00 00 00 00 lea    0x0(,%eax,4),%edx
//edx=4*num2
8049820:   8b 45 08             mov    0x8(%ebp),%eax
//eax=0xffffd33c
```

```
8049823:   01 d0                 add    %edx,%eax
//eax+=edx
8049825:   8b 00                 mov    (%eax),%eax
//eax=*eax
8049827:   eb 3c                 jmp    8049865 <func4+0x75>
//离开递归 return 0xffffd33c+4*num2
```

当 num1>num2 时，查询 0x8(%ebp)地址可以得到 0xffffd33c，分析可得 num1>num2 时跳转出递归，且 return 0xffffd33c+4*num2，可以猜测 0xffffd33c 内存放的是一个数组，并且 return 此数组一个指定值，且该指定地址由 num2 来确定。

综合分析以后，可以解析 func4 的函数如下：

```
int f(int a,int b)
{
    int c = (a + b) / 2;
    if (a < 0)
        return 0;
    if (a >= b)
    {
        int e = g[b];
        return e;
    }
    else if (f(a, c) > f(c + 1, b))
        return f(a, c);
    else
        return f(c + 1, b);
}
```

Gdb 进入 func4 内查询数组 0xffffd33c 存放的数值可以得到：

```
(gdb) x/100dw 0xffffd33c
0xffffd33c:     324     272     63      246
0xffffd34c:     73      404     369     387
0xffffd35c:     24      501     34      225
0xffffd36c:     254     78      341     135
0xffffd37c:     506     124     255     45
0xffffd38c:     322     223     309     244
0xffffd39c:     489     92      304     53
0xffffd3ac:     218     4       392     71
0xffffd3bc:     47      276     378     54
0xffffd3cc:     224     409     18      17
0xffffd3dc:     248     99      313     240
0xffffd3ec:     362     461     466     296
0xffffd3fc:     343     141     23      140
0xffffd40c:     1       228     11      177
0xffffd41c:     -687152128      134534144       1101775613      -11192
0xffffd42c:     -11136  -134467584      -134467584      -11160
0xffffd43c:     134517992       134534144       1048576 -11160
0xffffd44c:     134517978       1       -10988  -10980
0xffffd45c:     134534144       -11136  0       0
0xffffd46c:     -136335643      -134467584      -134467584      0
0xffffd47c:     -136335643      1       -10988  -10980
0xffffd48c:     -11100  -134467584      0       -11016
0xffffd49c:     0       -134230016      0       -134467584
0xffffd4ac:     -134467584      0       279577635       1348354611
0xffffd4bc:     0       0       0       1
(gdb)
```

为方便计算首先取得前 56 值并计入数组中，若获取不到准确数值则再加入后续数值。

为了获得符合 phase4 的数值，编写 cpp 代码如下：

```
#include<iostream>
using namespace std;
int g[56] =
{324,272,63,246,73,404,369,387,24,501,34,225,254,78,341,135,506,124,225,45,322,223,309,244,489,92,304,53,
218,4,392,71,47,276,378,54,224,409,18,17,248,99,313,240,362,461,466,296,343,141,23,140,1,228,11,177 };

int f(int a,int b)
{
    int c = (a + b) / 2;
    if (a < 0)
        return 0;
    if (a >= b)
    {
        int e = g[b];
        return e;
    }
    else if (f(a, c) > f(c + 1, b))
        return f(a, c);
    else
        return f(c + 1, b);
```

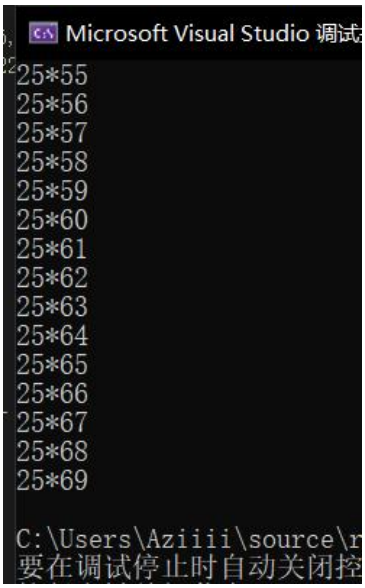
```
}

int main()
{
    for (int i = -5; i < 70; i++)
    {
        for (int j = -5; j < 70; j++)
        {
            if (f(i, j) == 466)
                if (f(i - 1, j) != 466 || i <= 0)
                    if (f(i, j + 1) != 466 || j > 54)
                    cout << i << "*" << j << endl;
        }
    }

    return 0;
}
```

其中

```
if (f(i, j) == 466)
                if (f(i - 1, j) != 466 || i <= 0)
                    if (f(i, j + 1) != 466 || j > 54)
                    cout << i << "*" << j << endl;
```

即为 phase4 中提到的三个接口，三种方式可以判定离开递归。

运行该函数，可以得到：



取得第一个值 25 66，运行代入 phase_4 中：

```
25 55
So you got that one.  Try this one.
```

Phase_5：

根据题目提示可得，本题需要输入一串字符串用于索引，不难猜想输入的字符串只是中间值，我们应该需要通过字符串的数值去查找所需要的东西，并进行最后的输出。

```
80499c4:   ff 75 08              pushl  0x8(%ebp)
//输入字符串
80499c7:   e8 68 04 00 00        call   8049e34 <string_length>
80499cc:   83 c4 10              add    $0x10,%esp
80499cf:   89 45 f4              mov    %eax,-0xc(%ebp)
80499d2:   83 7d f4 07           cmpl   $0x7,-0xc(%ebp)
//字符串长度=7
80499d6:   74 0c                 je     80499e4 <phase_5+0x2d>
80499d8:   e8 07 07 00 00        call   804a0e4 <explode_bomb>
```

分析前端代码可得，输入的字符串被保存再 0x8(%ebp)中，并且调用函数<string_length>查询字符串长度保存在 eax 中，最后与 7 相比较，如果不等于 7 则炸弹爆炸，说明需要输入的字符串长度为 7。

```
80499f4:   8b 55 ec              mov    -0x14(%ebp),%edx
//edx=k
80499f7:   8b 45 08              mov    0x8(%ebp),%eax
//eax=字符串
80499fa:   01 d0                 add    %edx,%eax
//eax=s[k]
```

通过该代码段分析得到，代码中加入一个参数-0x14(%ebp)，并且在 eax 存入输入的字符串，第三条代码使 eax 存入对应数组值，故将 edx 的内容暂且命名为计数器 k。

```
80499fc:   0f b6 00              movzbl (%eax),%eax
80499ff:   0f be c0              movsbl %al,%eax
8049a02:   83 e0 0f              and    $0xf,%eax
//字符对 16 取余

8049a05:   8b 04 85 60 d2 04 08  mov    0x804d260(,%eax,4),%eax
8049a0c:   01 45 f0              add    %eax,-0x10(%ebp)
//sum+=eax
8049a0f:   83 45 ec 01           addl   $0x1,-0x14(%ebp)
//k++
```

前半段代码将 eax 即之前选中的 string[k]中的值对 16 取余，并且用这个值调用 0x804d260 数组中保存的一个对应数值，将这个数值加入到新参数-0x10(%ebp)（记为 sum）中，之后之前的计数器自增。

```
8049a13:   83 7d ec 06           cmpl   $0x6,-0x14(%ebp)
//k<=6
8049a17:   7e db                 jle    80499f4 <phase_5+0x3d>
```

完成上步计算后 k 自增，判断 k 是否大于 6，若不大于则返回跳转到循环开始，即对新的 string[k]对应的数组值再次寻找并增加 sum，若大于 6 则说明字符串寻找完毕，跳出循环。

此处说明输入字符串长度确定为 7，且该循环是一个类似 for(int k=0;k<=6;k++)的循环。

```
8049a19:   83 7d f0 3c           cmpl   $0x3c,-0x10(%ebp)
//sum=60
```

```
8049a1d:   74 0c                 je     8049a2b <phase_5+0x74>
8049a1f:   e8 c0 06 00 00        call   804a0e4 <explode_bomb>
```

上述 for 循环结束后，判定 sum 是否等于 **0x3c**，转化为十进制的 60，即需要 sum=60，否则炸弹爆炸。

所以本题需要我们输入一个定值，以寻找对应数组 **0x804d260** 中所表示的值，并将这个值加入到 sum 之中，最终使得 7 位数加起来等于 60。

在 gdb 中查询 0x804d260 即可得到数组：

```
(gdb) x/20dw 0x804d260
0x804d260 <array.2714>: 4       14      16      5
0x804d270 <array.2714+16>:      12      3       8       15
0x804d280 <array.2714+32>:      2       13      10      6
0x804d290 <array.2714+48>:      11      7       1       9
0x804d2a0 <stdin@@GLIBC_2.0>:   0       0       0       0
(gdb)
```

为了使得 7 个数总和等于 60，我们选取 **3 7 10 10 10 10 10** 这 7 个数（选取情况不唯一），其对应的数组下标就是 **5 13 10 10 10 10 10**，则说明输入的字符串的 ASCII 码取余后等于 5 13 10 10 10 10 10，则对应 **e m j j j j j**。

将 emjjjjj 运行代入 phase_5 中得到：

```
emjjjjj
Good work!  On to the next...
```

注意：将 emjjjjj 乱序同样也可以，例如 ejjjjjm、mjejjjj。

Phase_6：

根据题目要求，本题需要输入一个字符串用于调整链表或者二叉树使得其对应所需要的顺序，所以此题基本与上题一致，通过输入一个字符串寻找其对应的数字并将其重新排序至所需顺序。

```
8049a57:   6a 07                 push   $0x7
```

```
//number=7
8049a59:   8d 45 bc              lea    -0x44(%ebp),%eax
8049a5c:   50                    push   %eax
8049a5d:   ff 75 a4              pushl  -0x5c(%ebp)
8049a60:   e8 3d 03 00 00        call   8049da2 <read_n_numbers>
//读入 7 个数字
```

分析前端代码可得，phase 需要输入长度为 7 的七个数字，并将这七个数字放入-0x5c(%ebp)中，且同时压入一个数组-0x44(%ebp)，故此处可确定答案长度。

```
8049a76:   c7 45 b0 00 00 00 00  movl   $0x0,-0x50(%ebp)
//-0x50(%ebp)循环变量 i
8049a7d:   eb 60                 jmp    8049adf <phase_6+0xad>

8049a7f:   8b 45 b0              mov    -0x50(%ebp),%eax
//循环开始 eax=i
8049a82:   8b 44 85 bc           mov    -0x44(%ebp,%eax,4),%eax
//eax=a[i]
8049a86:   85 c0                 test   %eax,%eax
8049a88:   7e 0c                 jle    8049a96 <phase_6+0x64>

8049a8a:   8b 45 b0              mov    -0x50(%ebp),%eax
//eax=i
8049a8d:   8b 44 85 bc           mov    -0x44(%ebp,%eax,4),%eax
//eax=[i]
8049a91:   83 f8 07              cmp    $0x7,%eax
//check(eax<=7)
8049a94:   7e 0f                 jle    8049aa5 <phase_6+0x73>
8049a96:   e8 49 06 00 00        call   804a0e4 <explode_bomb>

8049a9b:   b8 00 00 00 00        mov    $0x0,%eax
8049aa0:   e9 08 01 00 00        jmp    8049bad <phase_6+0x17b>
//离开循环

8049aa5:   8b 45 b0              mov    -0x50(%ebp),%eax
//eax=i
8049aa8:   83 c0 01              add    $0x1,%eax
//i++
8049aab:   89 45 b4              mov    %eax,-0x4c(%ebp)
//j=-0x4c(%ebp)=i+1
8049aae:   eb 25                 jmp    8049ad5 <phase_6+0xa3>
```

```
8049ab0:   8b 45 b0             mov    -0x50(%ebp),%eax
//eax=i
8049ab3:   8b 54 85 bc          mov    -0x44(%ebp,%eax,4),%edx
//edx=a[i]
8049ab7:   8b 45 b4             mov    -0x4c(%ebp),%eax
//eax=j
8049aba:   8b 44 85 bc          mov    -0x44(%ebp,%eax,4),%eax
8049abe:   39 c2                cmp    %eax,%edx
//a[i]=a[j]
8049ac0:   75 0f                jne    8049ad1 <phase_6+0x9f>
8049ac2:   e8 1d 06 00 00       call   804a0e4 <explode_bomb>
8049ac7:   b8 00 00 00 00       mov    $0x0,%eax
8049acc:   e9 dc 00 00 00       jmp    8049bad <phase_6+0x17b>

8049ad1:   83 45 b4 01          addl   $0x1,-0x4c(%ebp)
//j++
8049ad5:   83 7d b4 06          cmpl   $0x6,-0x4c(%ebp)
//j<=6
8049ad9:   7e d5                jle    8049ab0 <phase_6+0x7e>

8049adb:   83 45 b0 01          addl   $0x1,-0x50(%ebp)
//i++
8049adf:   83 7d b0 06          cmpl   $0x6,-0x50(%ebp)
//i<=6
8049ae3:   7e 9a                jle    8049a7f <phase_6+0x4d>
```

分析这段循环代码可得，此处代码为检查数组 a 中存放的值互不相等，其结构类同于：

```
for(int i=0;i<=6;i++)
{
  for(int j=0;j<=6;j++)
   { a[i]!=a[j];}
}
```

```
8049ae5:   c7 45 b0 00 00 00 00  movl   $0x0,-0x50(%ebp)
//i=0
8049aec:   eb 36                jmp    8049b24 <phase_6+0xf2>

8049aee:   8b 45 b8             mov    -0x48(%ebp),%eax
//eax=-0x48(%ebp)
8049af1:   89 45 ac             mov    %eax,-0x54(%ebp)
```

```
//p=eax
8049af4:   c7 45 b4 01 00 00 00  movl   $0x1,-0x4c(%ebp)
//j=1
8049afb:   eb 0d                 jmp    8049b0a <phase_6+0xd8>

8049afd:   8b 45 ac              mov    -0x54(%ebp),%eax
//eax=p
8049b00:   8b 40 08              mov    0x8(%eax),%eax
8049b03:   89 45 ac              mov    %eax,-0x54(%ebp)
//p=0x8(%eax)
8049b06:   83 45 b4 01           addl   $0x1,-0x4c(%ebp)

8049b0a:   8b 45 b0              mov    -0x50(%ebp),%eax
//eax=i
8049b0d:   8b 44 85 bc           mov    -0x44(%ebp,%eax,4),%eax
//eax=a[i]
8049b11:   39 45 b4              cmp    %eax,-0x4c(%ebp)
//j<a[i]
8049b14:   7c e7                 jl     8049afd <phase_6+0xcb>

8049b16:   8b 45 b0              mov    -0x50(%ebp),%eax
//eax=i
8049b19:   8b 55 ac              mov    -0x54(%ebp),%edx
//edx=p
8049b1c:   89 54 85 d8           mov    %edx,-0x28(%ebp,%eax,4)
8049b20:   83 45 b0 01           addl   $0x1,-0x50(%ebp)
//i++

8049b24:   83 7d b0 06           cmpl   $0x6,-0x50(%ebp)
//i<=6
8049b28:   7e c4                 jle    8049aee <phase_6+0xbc>
```

```
8049ae5:   c7 45 b0 00 00 00 00  movl   $0x0,-0x50(%ebp)
//i=0
8049aec:   eb 36                 jmp    8049b24 <phase_6+0xf2>
8049aee:   8b 45 b8              mov    -0x48(%ebp),%eax
//eax=-0x48(%ebp)
8049af1:   89 45 ac              mov    %eax,-0x54(%ebp)
//p=eax
8049af4:   c7 45 b4 01 00 00 00  movl   $0x1,-0x4c(%ebp)
//j=1
8049afb:   eb 0d                 jmp    8049b0a <phase_6+0xd8>

8049afd:   8b 45 ac              mov    -0x54(%ebp),%eax
```

```
//eax=p
8049b00:   8b 40 08              mov    0x8(%eax),%eax
8049b03:   89 45 ac              mov    %eax,-0x54(%ebp)
//p=0x8(%eax)
8049b06:   83 45 b4 01           addl   $0x1,-0x4c(%ebp)
//j++

8049b0a:   8b 45 b0              mov    -0x50(%ebp),%eax
//eax=i
8049b0d:   8b 44 85 bc           mov    -0x44(%ebp,%eax,4),%eax
//eax=a[i]
8049b11:   39 45 b4              cmp    %eax,-0x4c(%ebp)
//j<a[i]
8049b14:   7c e7                 jl     8049afd <phase_6+0xcb>

8049b16:   8b 45 b0              mov    -0x50(%ebp),%eax
//eax=i
8049b19:   8b 55 ac              mov    -0x54(%ebp),%edx
//edx=p
8049b1c:   89 54 85 d8           mov    %edx,-0x28(%ebp,%eax,4)
8049b20:   83 45 b0 01           addl   $0x1,-0x50(%ebp)
//i++

8049b24:   83 7d b0 06           cmpl   $0x6,-0x50(%ebp)
//i<=6
8049b28:   7e c4                 jle    8049aee <phase_6+0xbc>
```

分析这段代码可得，这段代码新开数组 b[i]，按照数组 a 中的顺序存入 a 数组所对应的链表的值。

```
8049b2a:   8b 45 d8              mov    -0x28(%ebp),%eax
//ax=b[0]
8049b2d:   89 45 b8              mov    %eax,-0x48(%ebp)
//bp-0x48=b[0]
8049b30:   8b 45 b8              mov    -0x48(%ebp),%eax
//eax=b[0]
8049b33:   89 45 ac              mov    %eax,-0x54(%ebp)
//bp-0x54=b[0]

8049b36:   c7 45 b0 01 00 00 00  movl   $0x1,-0x50(%ebp)
//i=1
8049b3d:   eb 1a                 jmp    8049b59 <phase_6+0x127>
```

```
8049b3f:   8b 45 b0             mov    -0x50(%ebp),%eax
//eax=i
8049b42:   8b 54 85 d8          mov    -0x28(%ebp,%eax,4),%edx
//dx=b[i]
8049b46:   8b 45 ac             mov    -0x54(%ebp),%eax
//eax=bp-0x54
8049b49:   89 50 08             mov    %edx,0x8(%eax)
//bp-0x54+8=b[i]
8049b4c:   8b 45 ac             mov    -0x54(%ebp),%eax
//ax=bp-0x54
8049b4f:   8b 40 08             mov    0x8(%eax),%eax
//ax=bp-0x54+8
8049b52:   89 45 ac             mov    %eax,-0x54(%ebp)
//bp-0x54=bx0x54+8
8049b55:   83 45 b0 01          addl   $0x1,-0x50(%ebp)
//i++

8049b59:   83 7d b0 06          cmpl   $0x6,-0x50(%ebp)
//i<=6
8049b5d:   7e e0                jle    8049b3f <phase_6+0x10d>
```

分析这段代码可得，这段代码将原链表按照数组 b 所存的值的顺序重新排序。

```
8049b5f:   8b 45 ac             mov    -0x54(%ebp),%eax
//ax=bp-0x54
8049b62:   c7 40 08 00 00 00 00 movl   $0x0,0x8(%eax)
//bp-0x54+8=0
8049b69:   8b 45 b8             mov    -0x48(%ebp),%eax
//ax=c[0]
8049b6c:   89 45 ac             mov    %eax,-0x54(%ebp)
//bp-0x54=c[0]

8049b6f:   c7 45 b0 00 00 00 00 movl   $0x0,-0x50(%ebp)
//i=0
8049b76:   eb 2a                jmp    8049ba2 <phase_6+0x170>

8049b78:   8b 45 ac             mov    -0x54(%ebp),%eax
//ax=bp-0x54
8049b7b:   8b 10                mov    (%eax),%edx
//dx=ax
8049b7d:   8b 45 ac             mov    -0x54(%ebp),%eax
8049b80:   8b 40 08             mov    0x8(%eax),%eax
```

```
//ax=*(bp-0x54)+8
8049b83:   8b 00                 mov    (%eax),%eax
//ax*(=*(bp-0x54)+8)
8049b85:   39 c2                 cmp    %eax,%edx
//ax=dx
8049b87:   7e 0c                 jle    8049b95 <phase_6+0x163>
8049b89:   e8 56 05 00 00        call   804a0e4 <explode_bomb>
8049b8e:   b8 00 00 00 00        mov    $0x0,%eax
8049b93:   eb 18                 jmp    8049bad <phase_6+0x17b>

8049b95:   8b 45 ac              mov    -0x54(%ebp),%eax
//ax=bp-0x54
8049b98:   8b 40 08              mov    0x8(%eax),%eax
//ax=*(bp-0x54)+8
8049b9b:   89 45 ac              mov    %eax,-0x54(%ebp)
//bp-0x54=*(bp-0x54)+8
8049b9e:   83 45 b0 01           addl   $0x1,-0x50(%ebp)
//i++

8049ba2:   83 7d b0 05           cmpl   $0x5,-0x50(%ebp)
//i<=5
8049ba6:   7e d0                 jle    8049b78 <phase_6+0x146>
8049ba8:   b8 01 00 00 00        mov    $0x1,%eax
```

分析这段代码可得，该链表需要以从小到大的顺序排序，否则炸弹爆炸，所以要求用户输入一组值重新排序该链表，使得链表所对应的值从小到大。

根据对出现的唯一明码进行持续检索可以得到：

```
(gdb) x/3d 0x804d198
0x804d198 <node1>:      1       0       0
(gdb) x/3w 0x804d198
0x804d198 <node1>:      1       1       134533516
(gdb) x/3w 134533516
0x804d18c <node2>:      2       2       134533504
(gdb) x/3w 134533504
0x804d180 <node3>:      8       3       134533492
(gdb) x/3w 134533492
0x804d174 <node4>:      7       4       134533480
(gdb) x/3w 134533480
0x804d168 <node5>:      0       5       134533468
(gdb) x/3w 134533468
0x804d15c <node6>:      9       6       134533456
(gdb) x/3w 134533456
0x804d150 <node7>:      4       7       0
(gdb)
```

两组排序可得：

A 1 2 8 7 0 9 4

B 1 2 3 4 5 6 7

对 A 组由小到大重新排序得：

A 0 1 2 4 7 8 9

B 5 1 2 7 4 3 6

则代入 5 1 2 7 4 3 6 运行 phase_6 可得：

```
5 1 2 7 4 3 6
Congratulations! You've defused the bomb!
```

secret_phase：

在 phase6 之后发现了<fun7>和<secret_phase>，说明 bomb 中还有一个隐藏炸弹，全局 Ctrl+f 搜索 secret_phase 后发现在<phase_defused>中调用了这个隐藏函数，所以进入 secret_phase 的方法在 phase_defused 中。

```
804a18f:   83 c4 10             add    $0x10,%esp
804a192:   e8 90 fa ff ff       call   8049c27 <secret_phase>
```

要到达调用该函数的位置必须通过之前的判断语句

```
804a126:   a1 ac d2 04 08        mov    0x804d2ac,%eax
804a12b:   83 f8 07              cmp    $0x7,%eax
804a12e:   75 77                 jne    804a1a7 <phase_defused+0x96>
```

说明必须 0x804d2ac 内存放七位字符串才能进入 secret_phase，并且通过查看之后的明码能发现进入的密码。

```
(gdb) x/s 0x804b406
0x804b406:      "%d %d %s"
(gdb) x/s 0x804d400
0x804d400 <input_strings+320>:  ""
(gdb) x/s 0x804b40f
0x804b40f:      "gxljebnA"
(gdb) x/s 0x804b418
0x804b418:      "Curses, you've found the secret phase!"
(gdb) x/s 0x804b478
0x804b478:      "Congratulations! You've defused the bomb!"
```

而且我们可以发现 0x804d400 该地址为空，说明这是需要我们填入密码的地方，通过对全局炸弹逐一分析，可以发现在 Phase4 结束后修改了 0x804d400 的内容，说明需要在 Phase4 答案后写入 gxljebnA 才可以进入 secret_phase。

```
8049c47:   89 45 f0              mov    %eax,-0x10(%ebp)
8049c4a:   83 7d f0 00           cmpl   $0x0,-0x10(%ebp)
8049c4e:   7e 09                 jle    8049c59 <secret_phase+0x32>
//>0

8049c50:   81 7d f0 e9 03 00 00  cmpl   $0x3e9,-0x10(%ebp)
8049c57:   7e 0c                 jle    8049c65 <secret_phase+0x3e>
//<=1001
```

查看 secert_phase 内容发现该 Phase 要求我们输入一个大于 0 且小于等于 1001 的数字。

```
8049c6b:   68 4c d2 04 08        push   $0x804d24c
8049c70:   e8 4b ff ff ff        call   8049bc0 <fun7>
```

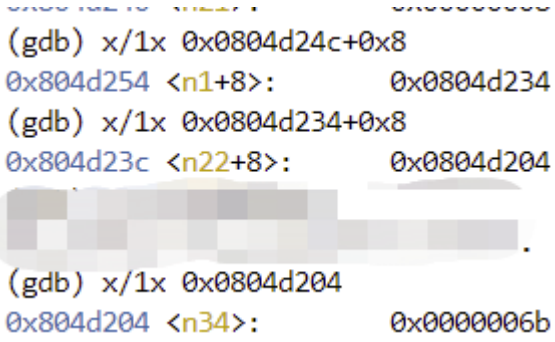
```
8049c75:   83 c4 10              add    $0x10,%esp
8049c78:   89 45 f4              mov    %eax,-0xc(%ebp)
8049c7b:   83 7d f4 03           cmpl   $0x3,-0xc(%ebp)
//=3
8049c7f:   74 0c                 je     8049c8d <secret_phase+0x66>
```

再往后发现该输入值和 0x804d24c 一起被调用函数 fun7，并且 fun7 返回值为 3 时跳过炸弹。

通过对 fun7 解析可得伪代码：

```
int fun7(int a,int b)
{
    if(a)
    {
        if(b>=a)
        {
            if(b==a)
            return 0;

            else
            return 2*fun7(*(a+8),b)+1;
        }
        else
        return 2*fun7(*(a+4),b);
    }
    else
    return -1;
 }
```

由于代码返回值为 3，说明要求 fun7(*(a+8),b)=1，而该值=1 说明上一次 fun7(*(a+8),b)=0，即*(*(a+8)+8)==b。



即可查出该位置存放 6b，转化为十进制 107。

将 107 代入运行可得

```
But finding it and solving it are quite different...
107
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

拆弹结束，完结撒花 XD