phase_2

```
08048c03 <phase_2>:
 8048c03:   55                      push   %ebp
 8048c04:   57                      push   %edi
 8048c05:   56                      push   %esi
 8048c06:   53                      push   %ebx
 8048c07:   83 ec 34                sub    $0x34,%esp
 8048c0a:   8d 5c 24 10             lea    0x10(%esp),%ebx
 8048c0e:   53                      push   %ebx
 8048c0f:   ff 74 24 4c             pushl  0x4c(%esp)
 8048c13:   e8 34 0b 00 00          call   804974c <read_six_numbers>
 8048c18:   8d 6c 24 24             lea    0x24(%esp),%ebp
 8048c1c:   83 c4 10                add    $0x10,%esp
 8048c1f:   be 00 00 00 00          mov    $0x0,%esi
 8048c24:   89 df                   mov    %ebx,%edi
 8048c26:   8b 43 0c                mov    0xc(%ebx),%eax
 8048c29:   39 03                   cmp    %eax,(%ebx)
 8048c2b:   74 05                   je     8048c32 <phase_2+0x2f>
 8048c2d:   e8 dd 0a 00 00          call   804970f <explode_bomb>
 8048c32:   03 37                   add    (%edi),%esi
 8048c34:   83 c3 04                add    $0x4,%ebx
 8048c37:   39 eb                   cmp    %ebp,%ebx
 8048c39:   75 e9                   jne    8048c24 <phase_2+0x21>
 8048c3b:   85 f6                   test   %esi,%esi
 8048c3d:   75 05                   jne    8048c44 <phase_2+0x41>
 8048c3f:   e8 cb 0a 00 00          call   804970f <explode_bomb>
 8048c44:   83 c4 2c                add    $0x2c,%esp
 8048c47:   5b                      pop    %ebx
 8048c48:   5e                      pop    %esi
 8048c49:   5f                      pop    %edi
 8048c4a:   5d                      pop    %ebp
 8048c4b:   c3                      ret
```

最前面 4 个 push 和最后 4 个 pop 是保护寄存器状态用的，可以无视。第一句起作用的语句是 `sub $0x34,%esp`，这句指令让栈指针自减 0x34，至于我们用到的数据在哪块内存，要继续往下看。后续函数压栈了两个地址，这里称它们为 addr1，addr2。然后调用 read_six_numbers 函数。简单看一看这个函数。

```
0804974c <read_six_numbers>:
 804974c:   83 ec 0c                sub    $0xc,%esp
 804974f:   8b 44 24 14             mov    0x14(%esp),%eax
 8049753:   8d 50 14                lea    0x14(%eax),%edx
 8049756:   52                      push   %edx
 8049757:   8d 50 10                lea    0x10(%eax),%edx
 804975a:   52                      push   %edx
 804975b:   8d 50 0c                lea    0xc(%eax),%edx
```

```
804975e:   52                      push   %edx
804975f:   8d 50 08                lea    0x8(%eax),%edx
8049762:   52                      push   %edx
8049763:   8d 50 04                lea    0x4(%eax),%edx
8049766:   52                      push   %edx
8049767:   50                      push   %eax
8049768:   68 cc 9e 04 08          push   $0x8049ecc
804976d:   ff 74 24 2c             pushl  0x2c(%esp)
8049771:   e8 5a f1 ff ff          call   80488d0 <__isoc99_sscanf@plt>
8049776:   83 c4 20                add    $0x20,%esp
8049779:   83 f8 05                cmp    $0x5,%eax
804977c:   7f 05                   jg     8049783 <read_six_numbers+0x37>
804977e:   e8 8c ff ff ff          call   804970f <explode_bomb>
8049783:   83 c4 0c                add    $0xc,%esp
8049786:   c3                      ret
```

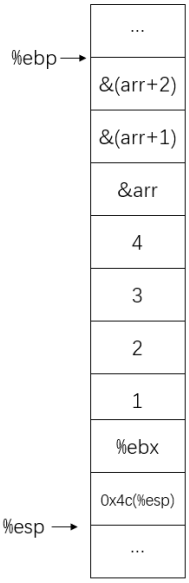
函数分配 3 个字，取出 addr1，然后把 mem[addr1:addr1+0x18]的内存范围内的这 6 个字的地址压栈，再压栈一个地址 0x8049ecc 和一个 0x2c(%esp)，调用 sscanf@plt；这是 linux 的 scanf 系统调用，用于读取用户的控制台输入。地址 0x8049ecc 应该指示了某种特殊的东西，gdb 调试过程中打印字符串

```
Breakpoint 1, 0x08048be2 in phase_1 ()
(gdb) x/s 0x8049ecc
0x8049ecc:      "%d %d %d %d %d %d"
```

易知这个 scanf 的格式化字符串是 6 个整形，那么第一关我们要输入的就是 6 个整形数字。这六个整形会被收集在
addr1 为基址的一个连续的 6 字长度的栈空间内。这样，我们认为这是一个 int arr[6]的连续数组。
在 scanf 返回后，函数回收栈空间，然后进行一个判断跳转逻辑；比较 scanf 的返回值和 5 的大小，如果 5 大于返回值则正常返回，小于等于返回值则触发炸弹。查 scanf 文档可知 scanf 的返回值是读到的参数数目，所以这一句是为了保证 scanf 顺利的读取到 6 个整形。这样，我们复刻出 read_six_numbers 函数的 c 语言形式

```c
void read_six_numbers(int arr[6]) {
    int ret = scanf("%d %d %d %d %d %d",arr,arr+1,arr+2,arr+3,arr+4,arr+5);
    if (ret<=5) explode_bomb();
    return ret;
}
```

读完了整形之后，回到 phase_2 函数继续运行；此时%ebx 寄存器中存储着也就是 arr 数组的基址地址&arr。
栈指针%esp 和调用 read_six_numbers 之前相同。栈的状态如下



此时 `lea 0x24(%esp),%ebp` 这一句让%ebp 指向&(arr+3)的位置。
然后程序让%esp+0x10，置%esi=0，%edi=%ebx，%eax=0xc(%ebx)=*(arr+3)=arr[3]。然后程序比较%eax 和(%ebx)，也就是比较 arr[0]和 arr[3]，如果不相等则爆炸。
避免了第一次爆炸后，寄存器%esi 增加(%edi)=arr[0]的数值，%ebx 加 4，再经过一层判断，决定 jmp 到低地址的指令，所以这应该是一个 do-while 循环语句。我们把每个寄存器视为局部变量，把 phase_2 翻译为 c 语言。

```
1.  void phase_2() {
2.      int arr[6];
3.      int *p = arr;
4.      read_six_number(p);
5.      int *bottom = arr+3;
6.      int i = 0;
7.      do {
8.          int *p2 = p;
9.          int *p3 = p+3;
10.         if (*p2!=*p3) explode_bomb();
11.         i += (*p2);
12.         p += 1;
13.     } while(bottom!=p)
14.     if (!i) explode_bomb();
15. }
```

这样，结论就很显而易见了。这个循环会执行三次，每次循环中会判断两个指针指向的数据是否相等，即分别判断 arr[0]?=arr[3]，arr[1]?=arr[4]，arr[2]?=arr[5]。在我们输入时，保证第一个数和第四个数相同，第二个数和第五个数相同，第三个数和第六个数相同即可。跳出循环后，还有一层判断，判断 i 是否为 0，这就增加了一层约束条件 arr[0]+ arr[1]+ arr[2] != 0。
同时满足两个条件的解很容易找到，如 1 2 3 1 2 3

phase_3

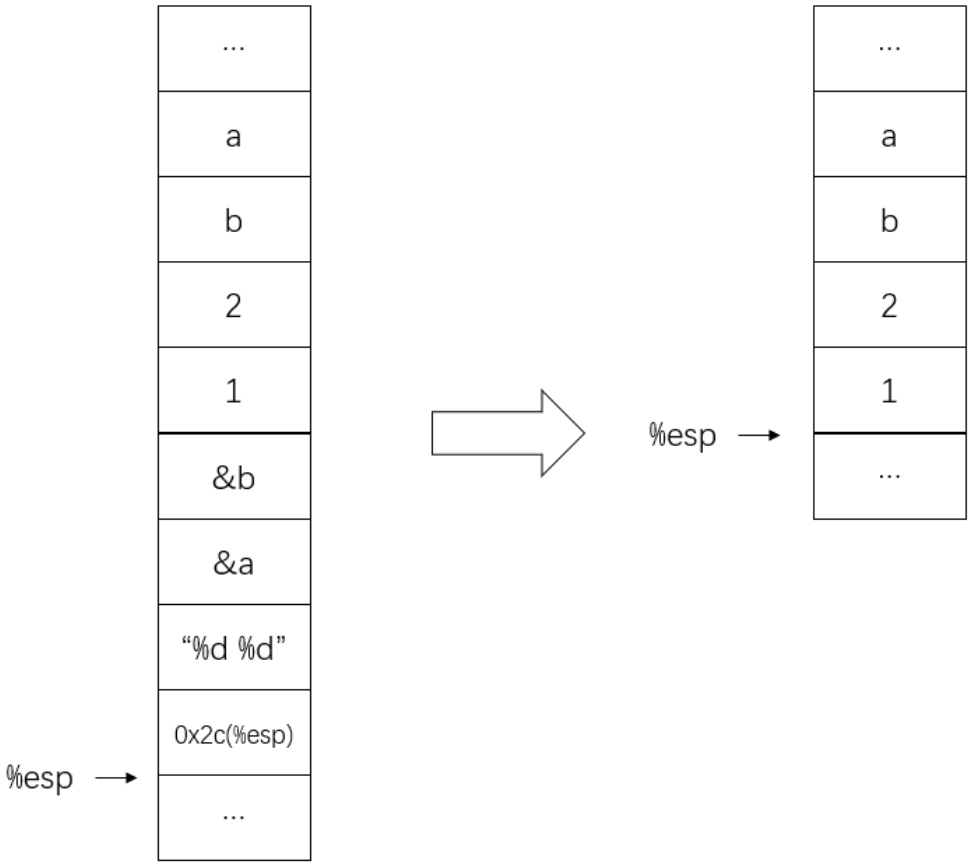
```
1.  08048c4c <phase_3>:
2.   8048c4c:   83 ec 1c               sub    $0x1c,%esp
```

```
3.  8048c4f:   8d 44 24 08            lea    0x8(%esp),%eax
4.  8048c53:   50                     push   %eax
5.  8048c54:   8d 44 24 10            lea    0x10(%esp),%eax
6.  8048c58:   50                     push   %eax
7.  8048c59:   68 d8 9e 04 08         push   $0x8049ed8
8.  8048c5e:   ff 74 24 2c            pushl  0x2c(%esp)
9.  8048c62:   e8 69 fc ff ff         call   80488d0 <__isoc99_sscanf@plt>
10. 8048c67:   83 c4 10               add    $0x10,%esp
11. 8048c6a:   83 f8 01               cmp    $0x1,%eax
12. 8048c6d:   7f 05                  jg     8048c74 <phase_3+0x28>
13. 8048c6f:   e8 9b 0a 00 00         call   804970f <explode_bomb>
14. 8048c74:   83 7c 24 0c 07         cmpl   $0x7,0xc(%esp)
15. 8048c79:   77 3c                  ja     8048cb7 <phase_3+0x6b>
16. 8048c7b:   8b 44 24 0c            mov    0xc(%esp),%eax
17. 8048c7f:   ff 24 85 40 9b 04 08   jmp    *0x8049b40(,%eax,4)
18. 8048c86:   b8 a0 00 00 00         mov    $0xa0,%eax
19. 8048c8b:   eb 3b                  jmp    8048cc8 <phase_3+0x7c>
20. 8048c8d:   b8 5f 01 00 00         mov    $0x15f,%eax
21. 8048c92:   eb 34                  jmp    8048cc8 <phase_3+0x7c>
22. 8048c94:   b8 d4 02 00 00         mov    $0x2d4,%eax
23. 8048c99:   eb 2d                  jmp    8048cc8 <phase_3+0x7c>
24. 8048c9b:   b8 b3 02 00 00         mov    $0x2b3,%eax
25. 8048ca0:   eb 26                  jmp    8048cc8 <phase_3+0x7c>
26. 8048ca2:   b8 3a 03 00 00         mov    $0x33a,%eax
27. 8048ca7:   eb 1f                  jmp    8048cc8 <phase_3+0x7c>
28. 8048ca9:   b8 b2 03 00 00         mov    $0x3b2,%eax
29. 8048cae:   eb 18                  jmp    8048cc8 <phase_3+0x7c>
30. 8048cb0:   b8 f9 00 00 00         mov    $0xf9,%eax
31. 8048cb5:   eb 11                  jmp    8048cc8 <phase_3+0x7c>
32. 8048cb7:   e8 53 0a 00 00         call   804970f <explode_bomb>
33. 8048cbc:   b8 00 00 00 00         mov    $0x0,%eax
34. 8048cc1:   eb 05                  jmp    8048cc8 <phase_3+0x7c>
35. 8048cc3:   b8 47 02 00 00         mov    $0x247,%eax
36. 8048cc8:   3b 44 24 08            cmp    0x8(%esp),%eax
37. 8048ccc:   74 05                  je     8048cd3 <phase_3+0x87>
38. 8048cce:   e8 3c 0a 00 00         call   804970f <explode_bomb>
39. 8048cd3:   83 c4 1c               add    $0x1c,%esp
40. 8048cd6:   c3                     ret
```

程序的开始仍然是分配 7 个字的栈空间，然后类似与前面的 read_six_numbers，压栈两个地址，一个格式化字符串，然后调用 scanf。我们还是先看看 0x8049ed8 里的格式化字符串是什么。

```
(gdb) x/s 0x8049ed8
0x8049ed8:      "%d %d"
```

是两个整形形式的元素，所以我们本关的输入是两个整形。调用 scanf 前后的栈状态为



这时 0xc(%esp)指示两个整数中的第一个，设为 a，0x8(%esp)指示第二个整数，设为 b。读取完用户输入后，程序首先判断(unsigned)(a)>7 ?，如果大于则触发 explode_bomb()，紧接着，使用一个带偏移的间接寻址的 jmp 语句，跳转到一个和 a 的数值直接相关的地址。为了看出 0x8049b40 这里究竟藏着什么，使用 gdb 查看

```
(gdb) x/10wx 0x8049b40
0x8049b40:      0x08048cc3      0x08048c86      0x08048c8d      0x08048c94
0x8049b50:      0x08048c9b      0x08048ca2      0x08048ca9      0x08048cb0
0x8049b60:      0x00000000      0x00000000
```

可以看到一个与 0x8049b40+4*(0~7)对应的地址表，那么这应该是一个 switch 语句。我们把 0~7 匹配到各个分支上，就能写出 phase_3 函数的 c 语言原型了。

```
1.  void phase_3() {
2.      int a, b;
3.      scanf("%d %d",&a, &b);
4.      if ((unsigned)a > 7) explode_bomb();
5.      int t = a;
6.      switch(t)
7.      {
8.      case 0:
9.        t = 0x247;
10.       break;
11.     case 1:
12.       t = 0xa0;
13.       break;
14.     case 2:
```

```
      t = 0x15f;
      break;
    case 3:
      t = 0x2d4;
      break;
    case 4:
      t = 0x2b3;
      break;
    case 5:
      t = 0x33a;
      break;
    case 6:
      t = 0x3b2;
      break;
    case 7:
      t = 0xf9;
      break;
    }
    if (b!=t) explode_bomb();
}
```

选择 a=0~7 的任一个值都能导向 switch 中的一条语句，为 t 赋予一个特定的值。这里我们选择 a=0，则 b=0x247。为了使判等成立，只需要令输入的 b=0x247 的十进制形式即可。

```
(gdb) print 0x247
$1 = 583
```

即一组可行解是 0 583

phase_4

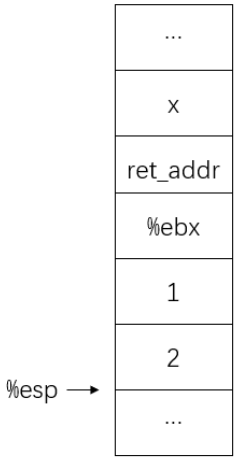
```
08048cd7 <func4>:
 8048cd7:   53                      push   %ebx
 8048cd8:   83 ec 08                sub    $0x8,%esp
 8048cdb:   8b 5c 24 10             mov    0x10(%esp),%ebx
 8048cdf:   b8 01 00 00 00          mov    $0x1,%eax
 8048ce4:   83 fb 01                cmp    $0x1,%ebx
 8048ce7:   7e 12                   jle    8048cfb <func4+0x24>
 8048ce9:   83 ec 0c                sub    $0xc,%esp
 8048cec:   8d 43 ff                lea    -0x1(%ebx),%eax
 8048cef:   50                      push   %eax
 8048cf0:   e8 e2 ff ff ff          call   8048cd7 <func4>
 8048cf5:   83 c4 10                add    $0x10,%esp
 8048cf8:   0f af c3                imul   %ebx,%eax
 8048cfb:   83 c4 08                add    $0x8,%esp
 8048cfe:   5b                      pop    %ebx
 8048cff:   c3                      ret

08048d00 <phase_4>:
 8048d00:   83 ec 20                sub    $0x20,%esp
```

```
8048d03:   8d 44 24 10           lea    0x10(%esp),%eax
8048d07:   50                    push   %eax
8048d08:   68 db 9e 04 08        push   $0x8049edb
8048d0d:   ff 74 24 2c           pushl  0x2c(%esp)
8048d11:   e8 ba fb ff ff        call   80488d0 <__isoc99_sscanf@plt>
8048d16:   83 c4 10              add    $0x10,%esp
8048d19:   83 f8 01              cmp    $0x1,%eax
8048d1c:   75 07                 jne    8048d25 <phase_4+0x25>
8048d1e:   83 7c 24 0c 00        cmpl   $0x0,0xc(%esp)
8048d23:   7f 05                 jg     8048d2a <phase_4+0x2a>
8048d25:   e8 e5 09 00 00        call   804970f <explode_bomb>
8048d2a:   83 ec 0c              sub    $0xc,%esp
8048d2d:   ff 74 24 18           pushl  0x18(%esp)
8048d31:   e8 a1 ff ff ff        call   8048cd7 <func4>
8048d36:   83 c4 10              add    $0x10,%esp
8048d39:   3d 80 9d 00 00        cmp    $0x9d80,%eax
8048d3e:   74 05                 je     8048d45 <phase_4+0x45>
8048d40:   e8 ca 09 00 00        call   804970f <explode_bomb>
8048d45:   83 c4 1c              add    $0x1c,%esp
8048d48:   c3                    ret
```

这一关有一个子函数 func4，一看就是和函数调用和递归相关的一关。
照例，我们看看 scanf 的输入格式

```
(gdb) x/s 0x8049edb
0x8049edb:      "%d"
```

只有一个整数，设它为 x。读取完毕后，程序调整%esp，这时 0xc(%esp)访问的是 x；比较 x 与 0，如果 x<=0 则触发 explode_bomb()，所以我们的输入必须是正整数。然后程序再次调整栈指针，将 x 的内容压栈，调用函数 func4。
让我们深究 func4 做了些什么。经过前几条语句的压栈，栈指针减后，栈状态为



所以此时 0x10(%esp)指向的是函数引用的参数 x。后续函数进行了一连串的条件判断和跳转，我们不妨先把它写成 c 语言，再分析函数究竟做了什么。

```c
int func4(int x) {
    if (x<=1) return 1;
    return x*func4(x-1);
}
```

考虑到 x 是正整数，所以 func4 起到的是递归求阶乘的作用。再回到 phase_4 函数，剩下的部分会判等返回值是否等于 0x9d80，如果不相等就触发炸弹。

```
1.  void phase_4() {
2.      int x;
3.      scanf("%d",&x);
4.      if (func4(x)!=0x9d80) explode_bomb();
5.  }
```

于是我们本关的输入 x 只需要满足 x!=0x9d80=40320 即可

```
(gdb) print 0x9d80
$2 = 40320
```

容易得到 x=8，所以答案是 8

phase_5

```
1.  08048d49 <phase_5>:
2.   8048d49:   83 ec 1c                sub    $0x1c,%esp
3.   8048d4c:   8d 44 24 08             lea    0x8(%esp),%eax
4.   8048d50:   50                      push   %eax
5.   8048d51:   8d 44 24 10             lea    0x10(%esp),%eax
6.   8048d55:   50                      push   %eax
7.   8048d56:   68 d8 9e 04 08          push   $0x8049ed8
8.   8048d5b:   ff 74 24 2c             pushl  0x2c(%esp)
9.   8048d5f:   e8 6c fb ff ff          call   80488d0 <__isoc99_sscanf@plt>
10.  8048d64:   83 c4 10                add    $0x10,%esp
11.  8048d67:   83 f8 01                cmp    $0x1,%eax
12.  8048d6a:   7f 05                   jg     8048d71 <phase_5+0x28>
13.  8048d6c:   e8 9e 09 00 00          call   804970f <explode_bomb>
14.  8048d71:   8b 44 24 0c             mov    0xc(%esp),%eax
15.  8048d75:   83 e0 0f                and    $0xf,%eax
16.  8048d78:   89 44 24 0c             mov    %eax,0xc(%esp)
17.  8048d7c:   83 f8 0f                cmp    $0xf,%eax
18.  8048d7f:   74 2e                   je     8048daf <phase_5+0x66>
19.  8048d81:   b9 00 00 00 00          mov    $0x0,%ecx
20.  8048d86:   ba 00 00 00 00          mov    $0x0,%edx
21.  8048d8b:   83 c2 01                add    $0x1,%edx
22.  8048d8e:   8b 04 85 80 9b 04 08    mov    0x8049b80(,%eax,4),%eax
23.  8048d95:   01 c1                   add    %eax,%ecx
24.  8048d97:   83 f8 0f                cmp    $0xf,%eax
25.  8048d9a:   75 ef                   jne    8048d8b <phase_5+0x42>
26.  8048d9c:   c7 44 24 0c 0f 00 00    movl   $0xf,0xc(%esp)
27.  8048da3:   00
28.  8048da4:   83 fa 09                cmp    $0x9,%edx
29.  8048da7:   75 06                   jne    8048daf <phase_5+0x66>
30.  8048da9:   3b 4c 24 08             cmp    0x8(%esp),%ecx
31.  8048dad:   74 05                   je     8048db4 <phase_5+0x6b>
32.  8048daf:   e8 5b 09 00 00          call   804970f <explode_bomb>
```

```
8048db4:   83 c4 1c                add    $0x1c,%esp
8048db7:   c3                      ret
```

本关的格式化字符串是

```
(gdb) x/s 0x8049ed8
0x8049ed8:      "%d %d"
```

是两个整形，其 scanf 读入部分和 phase_3 相同，因此同样的，0xc(%esp)指示两个整数中的第一个，设为 a，0x8(%esp)指示第二个整数，设为 b。然后，取 a 的低 4 位；后续涉及到一个循环，循环内会读一段连续内存(间隔 4 字节，说明是一个整形数组)，循环结束后还面临判断语句，我们把它翻译成 c 语言，再研究算法本质。

```c
extern int arr[16];
void phase_5() {
int a, b;
    scanf("%d %d",&a, &b);
    int i = a&0xf;
    a = i;
    if (i==15) explode_bomb();
    int c = 0, d = 0;
    do {
        d++;
        i = arr[i];
        c += i;
    } while(i!=0xf)
    a = 0xf;
    if (d!=9) explode_bomb();
    if (c!=b) explode_bomb();
}
```

首先，第一个 if 保证输入的 a 不能等于 15，否则炸弹直接爆炸。
然后进入循环，循环中我们不断读一个数组 arr 的某些下标位置，并把数组中的元素当成新的下标再次循环。用 gdb 查看 arr 数组内的元素都是些什么

```
(gdb) x/20wx 0x8049b80
0x8049b80 <array.2845>: 0x0000000a      0x00000002      0x0000000e      0x00000007
0x8049b90 <array.2845+16>:      0x00000008      0x0000000c      0x0000000f      0x0000000b
0x8049ba0 <array.2845+32>:      0x00000000      0x00000004      0x00000001      0x0000000d
0x8049bb0 <array.2845+48>:      0x00000003      0x00000009      0x00000006      0x00000005
0x8049bc0:      0x79206f53      0x7420756f      0x6b6e6968      0x756f7920
```

数组 arr = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}。
注意到循环的停止条件是下标为 15，不触发炸弹的条件是循环正好进行 9 次。则 i=15 的前一个下标 j 应该满足 arr[j]=15，也就是 j=7。为了得到 i=7 又可以继续前推，arr[j]=7，j=3…将这个过程重复 9 次即可，当然我们也不必手动重复，只需要写一段代码即可。

```
arr = [10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5]
iarr = [0 for _ in range(16)]

for i in range(16):
    iarr[arr[i]] = i

i = 15
print(15,end='')
for _ in range(9):
    i = iarr[i]
    print("->%d"%(i),end='')
```

executed in 12ms, finished 21:00:42 2021-02-15

```
15->6->14->2->1->10->0->8->4->9
```

可以得到，i 的初始值，也就是 a 应该等于 9。注意到退出循环后，程序还需要进行一次 b 和累加和的判等，这里的 b=4+8+0+10+1+2+14+6+15=60。
所以本题的解是 9 60

phase_6

```
08048e0d <phase_6>:
 8048e0d:   83 ec 10                sub    $0x10,%esp
 8048e10:   6a 0a                   push   $0xa
 8048e12:   6a 00                   push   $0x0
 8048e14:   ff 74 24 1c             pushl  0x1c(%esp)
 8048e18:   e8 33 fb ff ff          call   8048950 <strtol@plt>
 8048e1d:   a3 b4 b2 04 08          mov    %eax,0x804b2b4
 8048e22:   c7 04 24 b4 b2 04 08    movl   $0x804b2b4,(%esp)
 8048e29:   e8 8a ff ff ff          call   8048db8 <fun6>
 8048e2e:   8b 40 08                mov    0x8(%eax),%eax
 8048e31:   8b 40 08                mov    0x8(%eax),%eax
 8048e34:   8b 40 08                mov    0x8(%eax),%eax
 8048e37:   8b 40 08                mov    0x8(%eax),%eax
 8048e3a:   8b 40 08                mov    0x8(%eax),%eax
 8048e3d:   83 c4 10                add    $0x10,%esp
 8048e40:   8b 15 b4 b2 04 08       mov    0x804b2b4,%edx
 8048e46:   39 10                   cmp    %edx,(%eax)
 8048e48:   74 05                   je     8048e4f <phase_6+0x42>
 8048e4a:   e8 c0 08 00 00          call   804970f <explode_bomb>
 8048e4f:   83 c4 0c                add    $0xc,%esp
 8048e52:   c3                      ret
```

这一关的开局并不是 scanf 读输入，而是调用了 strtol 库函数。这个函数的功能可以查 cppReference 得到。

function

strtol <cstdlib>

```
long int strtol (const char* str, char** endptr, int base);
```

Convert string to long integer

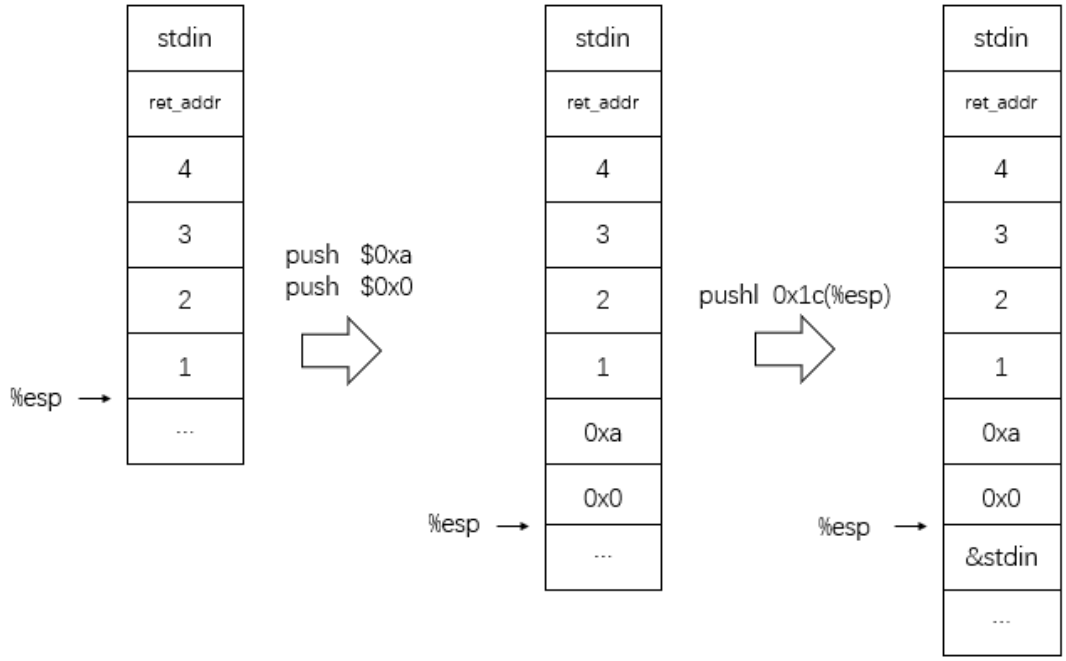
Parameters

`str`
C-string beginning with the representation of an integral number.

`endptr`
Reference to an object of type `char*`, whose value is set by the function to the next character in *str* after the numerical value.
This parameter can also be a null pointer, in which case it is not used.

`base`
Numerical base (radix) that determines the valid characters and their interpretation.
If this is `0`, the base used is determined by the format in the sequence (see above).

这个函数接收把 str 指向的字符串，按照 base 为基，转换为 long int 类型并返回。函数接收的 str 参数来自栈中，要知道它是什么，还需要精解一下栈的状态。

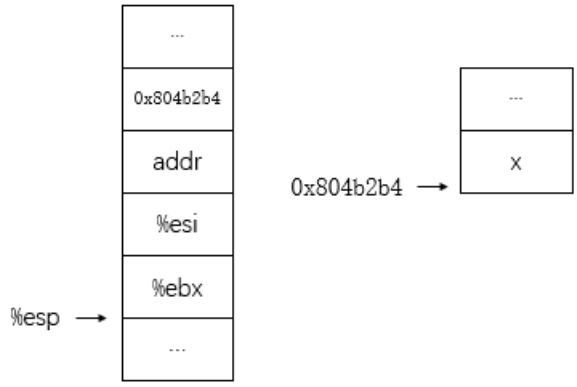


浅析前几条语句对应的栈状态后，可以发现传入 strtol 的 str 参数是 stdin 的栈内字符串，所以函数实际上是把控制台输入转换成整形。程序把这个整形存入 0x804b2b4 这个地址，再把 0x804b2b4 写入栈指针指向的地址，然后调用函数 fun6。

```
1.  08048db8 <fun6>:
2.    8048db8:   56                    push   %esi
3.    8048db9:   53                    push   %ebx
4.    8048dba:   8b 44 24 0c           mov    0xc(%esp),%eax
5.    8048dbe:   8b 70 08              mov    0x8(%eax),%esi
6.    8048dc1:   c7 40 08 00 00 00 00  movl   $0x0,0x8(%eax)
7.    8048dc8:   85 f6                 test   %esi,%esi
8.    8048dca:   75 2c                 jne    8048df8 <fun6+0x40>
9.    8048dcc:   eb 3c                 jmp    8048e0a <fun6+0x52>
10.   8048dce:   89 d1                 mov    %edx,%ecx
11.   8048dd0:   8b 51 08              mov    0x8(%ecx),%edx
12.   8048dd3:   85 d2                 test   %edx,%edx
13.   8048dd5:   74 04                 je     8048ddb <fun6+0x23>
14.   8048dd7:   39 1a                 cmp    %ebx,(%edx)
15.   8048dd9:   7f f3                 jg     8048dce <fun6+0x16>
16.   8048ddb:   39 d1                 cmp    %edx,%ecx
```

```
8048ddd:   74 05                 je     8048de4 <fun6+0x2c>
8048ddf:   89 71 08              mov    %esi,0x8(%ecx)
8048de2:   eb 08                 jmp    8048dec <fun6+0x34>
8048de4:   89 f0                 mov    %esi,%eax
8048de6:   eb 04                 jmp    8048dec <fun6+0x34>
8048de8:   89 c2                 mov    %eax,%edx
8048dea:   89 f0                 mov    %esi,%eax
8048dec:   8b 4e 08              mov    0x8(%esi),%ecx
8048def:   89 56 08              mov    %edx,0x8(%esi)
8048df2:   85 c9                 test   %ecx,%ecx
8048df4:   74 14                 je     8048e0a <fun6+0x52>
8048df6:   89 ce                 mov    %ecx,%esi
8048df8:   85 c0                 test   %eax,%eax
8048dfa:   74 ec                 je     8048de8 <fun6+0x30>
8048dfc:   8b 1e                 mov    (%esi),%ebx
8048dfe:   89 c1                 mov    %eax,%ecx
8048e00:   39 18                 cmp    %ebx,(%eax)
8048e02:   7f cc                 jg     8048dd0 <fun6+0x18>
8048e04:   89 c2                 mov    %eax,%edx
8048e06:   89 f0                 mov    %esi,%eax
8048e08:   eb e2                 jmp    8048dec <fun6+0x34>
8048e0a:   5b                    pop    %ebx
8048e0b:   5e                    pop    %esi
8048e0c:   c3                    ret
```

压栈操作完毕后，栈状态为



所以 0xc(%esp)引用刚写入栈内的地址 0x804b2b4，紧接着后面还使用了 0x804b2b4+8；这实在很可疑，使用 gdb 看一下 0x804b2b4 到底有什么。

```
(gdb) x/40wx 0x804b2b4
0x804b2b4 <node0>:      0x00000000      0x00000000      0x0804b2c0      0x00000075
0x804b2c4 <node1+4>:    0x00000001      0x0804b2cc      0x0000024f      0x00000002
0x804b2d4 <node2+8>:    0x0804b2d8      0x000001ef      0x00000003      0x0804b2e4
0x804b2e4 <node4>:      0x000002c9      0x00000004      0x0804b2f0      0x00000212
0x804b2f4 <node5+4>:    0x00000005      0x0804b2fc      0x00000116      0x00000006
0x804b304 <node6+8>:    0x0804b308      0x00000197      0x00000007      0x0804b314
0x804b314 <node8>:      0x0000039d      0x00000008      0x0804b320      0x0000034e
0x804b324 <node9+4>:    0x00000009      0x00000000      0x00000000      0x00000000
0x804b334:      0x00000000      0x00000000      0x00000000      0x35314553
```

有意思的要来了，可以看到在连续的一段地址上，存储着 10 个 node 类型，每个 node 使用了 3 个字的空间，而且每个字各有特点，我们有理由推断这是一个链表结点类型。

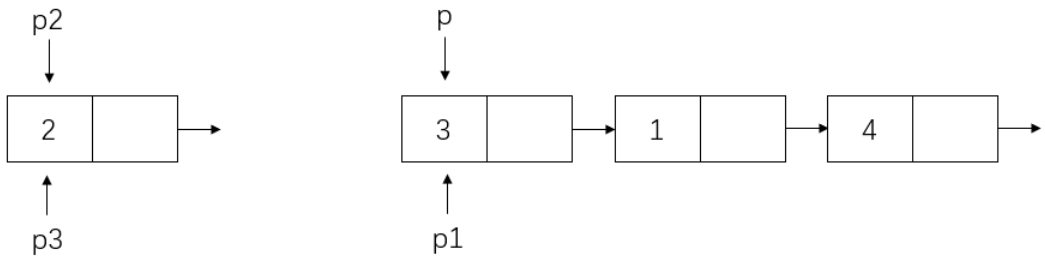
```
1.  struct node {
2.      int val;
3.      int id;
4.      node* next;
5.  };
```

这样，我们就能很清楚地理解代码中对地址的+4，+8 操作所代表的意义。
fun6 的代码涉及大量的条件分支跳转，而且是向前跳转，说明函数很可能含有复杂的循环。为了较好地剖析代码块，我们找到最大的循环，再逐层分析内部的循环。比较好的办法是先把代码分解成几个代码块，记录下每一个 jmp 指令的目标地址，把这些地址都定义成一个 while 循环的入口，并结合代码执行流程分析代码的流动线路。

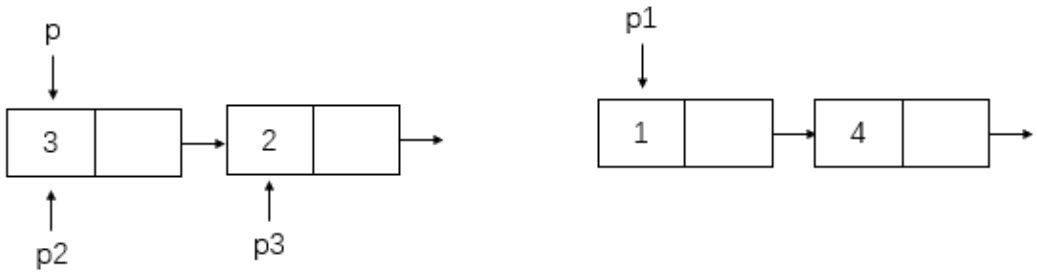
```
node* fun6(node* p) {
    /*
    p: %eax
    p1: %esi
    p1_val: %ebx
    p2: %ecx
    p3: %edx
    */
    node* p1 = p->next, p2, p3;
    p->next = 0;
    int p1_val;
    if (!p1) return p;
    go to entry;
    while (1) {
        while (1) {
            p2 = p1->next;
            p1->next = p3;
            if (!p2) return p;
            p1 = p2;
        entry:
            if (!p) {
                p3 = p;
                p = p1;
            }
            else {
                p1_val = p1->val;
                p2 = p;
```

```
                if ((p->val)>p1_val) break;
                p3 = p;
                p = p1;
            }
        }
        do {
            p3 = p2->next;
            if (p3 && (p3->val)>(p1_val)) {
                p2 = p3;
                continue;
            }
            else break;
        } while (1);
        if (p2==p3) {
            p = p1;
        }
        else {
            p2->next = p1;
        }
    }
}
```

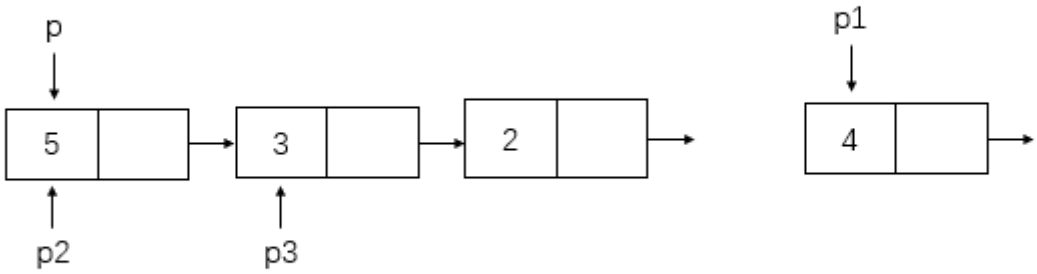
程序第一次进入 entry 后，会对 p1,p2,p3 指针做初始化赋值，这时的指针指向为



如果 p2->val 大于 p1->val，则跳出循环。否则，程序回到 loop1，把链表进行如下的调整

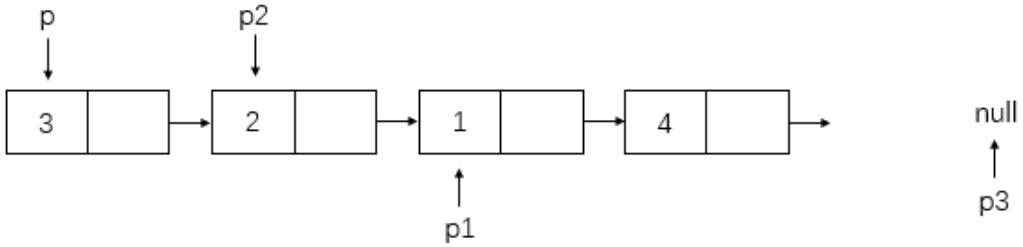


如果此时的 p2->val 仍大于 p1->val，则继续进行一次循环，调整链表

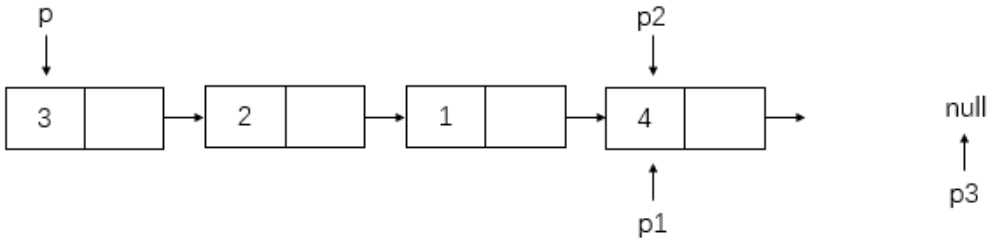


如果不大于则可以跳出循环。可以看出，如果 p2->val 小于等于 p1->val，则把它接到左侧链表 L1 的首部。如此循环下去，loop1 结束时，我们保证左侧链表 L1=p 是一个从首部往尾部降序的链表，且右侧链表 L2=p1=p2 的首部 p1->val 小于当前 L1 首部的元素。

进入 loop2，这个循环比较简单，它只改变 p2 和 p3。这个循环持续进行，直到 p3->val<=p1->val 或者 p3 为空。也就是说，p3 从 L1 的首部向后查找 L1，这个循环结束时，p3 指向 L1 中第一个小于等于 p1->val 的结点。然后程序进行把当前的 p2 和 p1 连接



再次回到 loop1，程序把 p1->next 当成新的 L2 首部，把 p1 当成新的 L1 尾部，继续迭代。



综上，程序的思路可以分为三个部分。
1. 把 L2 的首部的递减部分连接到 L1 的首部
2. 找到 L1 中第一个比 L2->val 小的结点 small，把 L2 插入到 L1 中合适的位置
3. 回到 1，继续迭代。

按照这个思路，我们可以复现出比上面更加易懂的 c 代码

```
node* fun6(node* L1) {
    node* L2 = L1->next;
    L1->next = 0;
```

```
    while (L2) {
        // 把L2的降序部分尽可能放到L1的首部
        while (L2 && L2->val>=L1->val) {
            node* tmp = L2->next;
            L2->next = L1;
            L1 = L2;
            L2 = tmp;
        }
        // 把L2的首部结点插入到合适的位置
        node *p = L2;
        L2 = L2->next;
        node* hole = L1, *hole_next = L1->next;
        while (hole_next && hole_next->val>p->val) {
            hole = hole_next;
            hole_next = hole->next;
        }
        hole->next = p;
        p->next = hole_next;
    }
    return L1;
}
```

容易看出，代码不断从 L2 的首部取元素插入到 L1，且 L1 始终降序排序；这说明这个函数的作用是把输入的链表 L1 用插入排序的方法降序排序，并返回链表头。

排序的 key 是结点的 val 元素。回到 phase_6，把 phase_6 也翻译成 c 代码。

```
extern node* p; // 链表头，程序中的0x804b2b4

void phase_6() {
    char str[100];
    scanf("%s",s);
    int val = strtol(str,0,10);
    p->val = val;
    node* head = fun6(p); // 排序
    head = head->next;
    head = head->next;
    head = head->next;
    head = head->next;
    head = head->next;
    if (head->val!=p->val) explode_bomb();
}
```

完成排序后，我们希望 0x804b2b4 出现在链表的第 6 位。为此，我们把 gdb 告诉我们的 9 个 node 用自己的高级语言工具排序。

```
sorted([0x75, 0x24f, 0x1ef, 0x2c9, 0x212, 0x116, 0x197, 0x39d, 0x34e])
```

executed in 10ms, finished 13:45:13 2021-02-16

```
[117, 278, 407, 495, 530, 591, 713, 846, 925]
```

所以只要输入的值满足 495<=x<=530 即可，因此一个可行答案是 530

phase_7

最后一关的输入格式和前一关相同，是输入一个整形，不再赘述。

```
08048ea4 <secret_phase>:
 8048ea4:   53                      push   %ebx
 8048ea5:   83 ec 08                sub    $0x8,%esp
 8048ea8:   e8 da 08 00 00          call   8049787 <read_line>
 8048ead:   83 ec 04                sub    $0x4,%esp
 8048eb0:   6a 0a                   push   $0xa
 8048eb2:   6a 00                   push   $0x0
 8048eb4:   50                      push   %eax
 8048eb5:   e8 96 fa ff ff          call   8048950 <strtol@plt>
 8048eba:   89 c3                   mov    %eax,%ebx
 8048ebc:   8d 40 ff                lea    -0x1(%eax),%eax
 8048ebf:   83 c4 10                add    $0x10,%esp
 8048ec2:   3d e8 03 00 00          cmp    $0x3e8,%eax
 8048ec7:   76 05                   jbe    8048ece <secret_phase+0x2a>
 8048ec9:   e8 41 08 00 00          call   804970f <explode_bomb>
 8048ece:   83 ec 08                sub    $0x8,%esp
 8048ed1:   53                      push   %ebx
 8048ed2:   68 00 b2 04 08          push   $0x804b200
 8048ed7:   e8 77 ff ff ff          call   8048e53 <fun7>
 8048edc:   83 c4 10                add    $0x10,%esp
 8048edf:   83 f8 06                cmp    $0x6,%eax
 8048ee2:   74 05                   je     8048ee9 <secret_phase+0x45>
 8048ee4:   e8 26 08 00 00          call   804970f <explode_bomb>
 8048ee9:   83 ec 0c                sub    $0xc,%esp
 8048eec:   68 f0 9a 04 08          push   $0x8049af0
 8048ef1:   e8 6a f9 ff ff          call   8048860 <puts@plt>
 8048ef6:   e8 53 09 00 00          call   804984e <phase_defused>
 8048efb:   83 c4 18                add    $0x18,%esp
 8048efe:   5b                      pop    %ebx
 8048eff:   c3                      ret
```

先翻译成 c 代码。

```
void phase_7() {
    char str[100];
    scanf("%s",s);
    int x = strtol(str,0,10);
    if ((unsigned)(x-1)>0x3e8) explode_bomb();
    int t = fun7(0x804b200, x);
    if (t!=6) explode_bomb();
}
```

程序限制 x 是小于等于 0x3e9 的正整数，然后压栈 x 和一个地址，调用函数 fun7.

```
08048e53 <fun7>:
 8048e53:   53                      push   %ebx
```

```
8048e54:   83 ec 08              sub    $0x8,%esp
8048e57:   8b 54 24 10           mov    0x10(%esp),%edx
8048e5b:   8b 4c 24 14           mov    0x14(%esp),%ecx
// %ebx=0x804b200, %ecx = x
8048e5f:   85 d2                 test   %edx,%edx
8048e61:   74 37                 je     8048e9a <fun7+0x47>
8048e63:   8b 1a                 mov    (%edx),%ebx
8048e65:   39 cb                 cmp    %ecx,%ebx
8048e67:   7e 13                 jle    8048e7c <fun7+0x29>
8048e69:   83 ec 08              sub    $0x8,%esp
8048e6c:   51                    push   %ecx
8048e6d:   ff 72 04              pushl  0x4(%edx)
8048e70:   e8 de ff ff ff        call   8048e53 <fun7>
8048e75:   83 c4 10              add    $0x10,%esp
8048e78:   01 c0                 add    %eax,%eax
8048e7a:   eb 23                 jmp    8048e9f <fun7+0x4c>
8048e7c:   b8 00 00 00 00        mov    $0x0,%eax
8048e81:   39 cb                 cmp    %ecx,%ebx
8048e83:   74 1a                 je     8048e9f <fun7+0x4c>
8048e85:   83 ec 08              sub    $0x8,%esp
8048e88:   51                    push   %ecx
8048e89:   ff 72 08              pushl  0x8(%edx)
8048e8c:   e8 c2 ff ff ff        call   8048e53 <fun7>
8048e91:   83 c4 10              add    $0x10,%esp
8048e94:   8d 44 00 01           lea    0x1(%eax,%eax,1),%eax
8048e98:   eb 05                 jmp    8048e9f <fun7+0x4c>
// return -1
8048e9a:   b8 ff ff ff ff        mov    $0xffffffff,%eax
8048e9f:   83 c4 08              add    $0x8,%esp
8048ea2:   5b                    pop    %ebx
8048ea3:   c3                    ret
```

照例 gdb 看一看 0x804b200 这个地址内都是什么。

```
(gdb) x/50wx 0x804b200
0x804b200 <n1>: 0x00000024      0x0804b20c      0x0804b218      0x00000008
0x804b210 <n21+4>:      0x0804b23c      0x0804b224      0x00000032      0x0804b230
0x804b220 <n22+8>:      0x0804b248      0x00000016      0x0804b290      0x0804b278
0x804b230 <n33>:        0x0000002d      0x0804b254      0x0804b29c      0x00000006
0x804b240 <n31+4>:      0x0804b260      0x0804b284      0x0000006b      0x0804b26c
0x804b250 <n34+8>:      0x0804b2a8      0x00000028      0x00000000      0x00000000
0x804b260 <n41>:        0x00000001      0x00000000      0x00000000      0x00000063
0x804b270 <n47+4>:      0x00000000      0x00000000      0x00000023      0x00000000
0x804b280 <n44+8>:      0x00000000      0x00000007      0x00000000      0x00000000
0x804b290 <n43>:        0x00000014      0x00000000      0x00000000      0x0000002f
0x804b2a0 <n46+4>:      0x00000000      0x00000000      0x000003e9      0x00000000
0x804b2b0 <n48+8>:      0x00000000      0x00000000      0x00000000      0x0804b2c0
0x804b2c0 <node1>:      0x00000075      0x00000001
```

名为 n 的结构体，推断是我们常用的二叉树结点类型

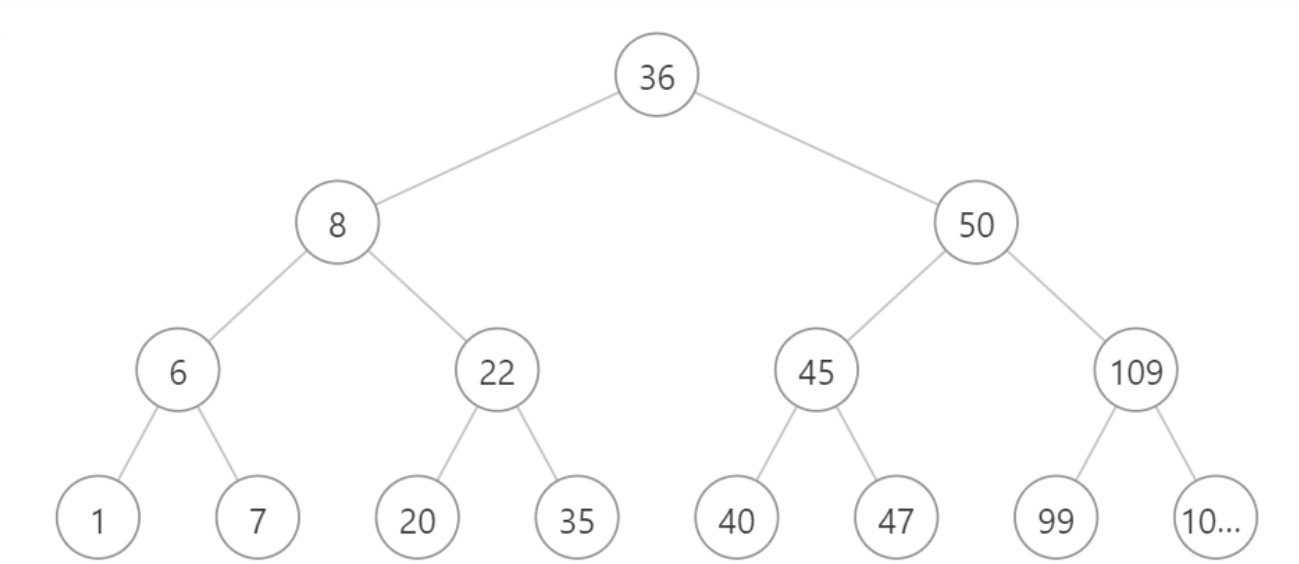
```
struct TreeNode* {
    int val;
    TreeNode* left;
```

```
4.      TreeNode* right;
5.  };
```

fun7 中包含递归调用，这个函数应该是对二叉树进行某种操作。我们详解这个函数并写出 c 代码

```c
int fun7(int* p, int x) {
    if (!p) return -1;
    int y = p->val;
    if (x<y) {
        return 2*fun7(p->left, x);
    }
    else if (x==y){
        return 0;
    }
    else {
        return 2*fun7(p->right, x)+1;
    }
}
```

是一个标准的递归搜索二叉树的函数。为了确定怎样的输入可以让 fun7 返回 6，我们有必要根据内存内的值还原二叉树的结构。



可以看出这是一个二叉搜索树。根据递归的计算式，我们很容易推导出

1. fun7(p, x) = 6 只能由 2*fun7(p->left, x)得到，且 fun7(p->left, x)=3。如果是由 2*fun7(p->right, x)+1 得到，则需要 fun7(p->right, x)=2.5，然而函数的返回值是整形，这个条件显然不成立。
2. fun7(p->left, x) = 3 只能由 2*fun7(p->left->right, x)+1 得到，且 fun7(p->left->right, x)=1。如果是由 2*fun7(p->left->left, x)得到，则需要 fun7(p->left->left, x)=1.5，然而函数的返回值是整形，这个条件显然不成立。
3. fun7(p->left->right, x)=1 只能由 2*fun7(p->left->right->right, x)+1 得到，且 fun7(p->left->right->right, x)=0。因为如果是由 2*fun7(p->left->right->left, x)得到，则 fun7(p->left->right->left, x)=0.5，然而函数的返回值是整形，这个条件显然不成立。
4. fun7(p->left->right->right, x)=0，使得 x=35，否则 fun7(p->left->right->right, x)= fun7(0, x)=-1。

所以我们导出了唯一答案，就是 x=35 时才能让返回值等于 6，所以本关答案是 35

。