Problem 1:

Consider the following C code:

```
int scrat(int val, int n)
{
  int result = 0;
  if(n > 0)
    result = val + scrat(val, n-1);
  return result;
}
```

A compiler on an IA-32 Linux machine produces the following object code for this function, which we have disassembled (using `objdump`) back into assembly code:

```
 08048390 <scrat>:
  8048390:       55                      push   %ebp
->8048391:       89 e5                   mov    %esp,%ebp
  8048393:       53                      push   %ebx
  8048394:       83 ec 08                sub    $0x8,%esp
  8048397:       8b 5d 08                mov    0x8(%ebp),%ebx
  804839a:       8b 45 0c                mov    0xc(%ebp),%eax
  804839d:       ba 00 00 00 00          mov    $0x0,%edx
  80483a2:       85 c0                   test   %eax,%eax
  80483a4:       7e 10                   jle    80483b6 <scrat+0x26>
  80483a6:       48                      dec    %eax
  80483a7:       89 44 24 04             mov    %eax,0x4(%esp,1)
  80483ab:       89 1c 24                mov    %ebx,(%esp,1)
  80483ae:       e8 dd ff ff ff          call   8048390 <scrat>
  80483b3:       8d 14 18                lea    (%eax,%ebx,1),%edx
  80483b6:       89 d0                   mov    %edx,%eax
  80483b8:       83 c4 08                add    $0x8,%esp
  80483bb:       5b                      pop    %ebx
  80483bc:       5d                      pop    %ebp
  80483bd:       c3                      ret
```

A. On the next page, we have the diagram of the stack immediately after some function makes a call to `scrat` and the very first instruction of `scrat` has executed (the next instruction to be executed is denoted with an arrow (->). The value of register `%esp` at this point is `0xbffff998`. For each of the numeric values shown in the table, give a short description of the value. If the value has a corresponding variable in the original C source code, use the name of the variable as its description.

B. Assume that `scrat` runs until it reaches the position denoted with an arrow (->) again. In the table, below the given numeric values, fill in the updated stack. Use a numeric value (if possible, else write `n/a`) and provide a short description of the value. Cross out any stack space not used.

Address	Numeric Value	Comments/Description
0xbffff9a4	0x00000003	n
0xbffff9a0	0x00000021	val
0xbffff99c	0x080483db	return address
0xbffff998	0xbffff9a8	stored ebp
0xbffff994	n/a	stored ebx
0xbffff990	0x00000002	n-1
0xbffff98c	0x00000021	val
0xbffff988	0x080483b3	return address
0xbffff984	0xbffff998	stored ebp
0xbffff980	----------	----------
0xbffff97c		
0xbffff978		
0xbffff974		
0xbffff970		
0xbffff97c		
0xbffff978		
0xbffff974		

Problem 2:

Consider the following source code and corresponding assembly:

```
#include <stdio.h>
char read_char()
{
    char c;
    scanf("%s",&c);
    return c;
}

int main()
{
    printf("%c\n", read_char());
}
```

```
080484c0 <read_char>:
 80484c0:       55                      push   %ebp
 80484c1:       89 e5                   mov    %esp,%ebp
 80484c3:       83 ec 10                sub    $0x10,%esp
 80484c6:       8d 45 ff                lea    0xffffffff(%ebp),%eax
 80484c9:       50                      push   %eax
 80484ca:       68 98 85 04 08          push   $0x8048598
 80484cf:       e8 6c fe ff ff          call   8048340 <_init+0x38>
 80484d4:       0f be 45 ff             movsbl 0xffffffff(%ebp),%eax
 80484d8:       89 ec                   mov    %ebp,%esp
 80484da:       5d                      pop    %ebp
 80484db:       c3                      ret
 80484dc:       8d 74 26 00             lea    0x0(%esi,1),%esi

080484e0 <main>:
 80484e0:       55                      push   %ebp
 80484e1:       89 e5                   mov    %esp,%ebp
 80484e3:       83 ec 08                sub    $0x8,%esp
 80484e6:       83 e4 f0                and    $0xfffffff0,%esp
 80484e9:       83 ec 10                sub    $0x10,%esp
 80484ec:       e8 cf ff ff ff          call   80484c0 <read_char>
 80484f1:       83 c4 08                add    $0x8,%esp
 80484f4:       0f be c0                movsbl %al,%eax
 80484f7:       50                      push   %eax
 80484f8:       68 9b 85 04 08          push   $0x804859b
 80484fd:       e8 6e fe ff ff          call   8048370 <_init+0x68>
 8048502:       89 ec                   mov    %ebp,%esp
 8048504:       5d                      pop    %ebp
 8048505:       c3                      ret
```

When the program is executed, a breakpoint is placed at the address 0x080484d4. The user inputs the single character 'a' (ASCII value 0x61). Assume that when the program is run the value of `%esp` immediately before the first instruction of `main` (0x080484e0) is executed is 0xbffff6ec. Assume that the entire stack space was initialized to zeroes before the program began executing and `main` is the first function to execute. Fill in the following table with the values observed to be on the stack when the program halts at the breakpoint. Clearly indicate (with labeled arrows) what the observed values of the registers `%ebp` and `%esp` would be. If the exact hexadecimal value cannot be determined from the information provided, but the value has a meaning (ie, return address, saved ebp), write the meaning. Your answers should be the 4-byte hexadecimal values at the stack location (**remember this is a little endian machine**. You do not need to write out all 8 digits of the hexadecimal number zero.

Address	Hex Value
0xbffff6ec	`return address`
0xbffff6e8	`saved ebp`
0xbffff6e4	0
0xbffff6e0	0
0xbffff6dc	0
0xbffff6d8	0
0xbffff6d4	0
0xbffff6d0	0
0xbffff6cc	`0x080484f1`
0xbffff6c8 `(ebp)`	`0xbffff600`
0xbffff6c4	`0x61000000`
0xbffff6c0	0
0xbffff6bc	0
0xbffff6b8	0
0xbffff6b4	`0xbffff6c7`
0xbffff6b0 `(esp)`	`0x08048598`
0xbffff6ac	
0xbffff6a8	
0xbffff6a4	
0xbffff6a0	