Does this code look okay?

```c
int binarySearch(int a[], int length, int key) {
    int low = 0;
    int high = length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid;  // key found
    }
    return -1;  // key not found
}
```
int binarySearch(int a[], int length, int key) {
    int low = 0;
    int high = length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid;  // key found
    }
    return -1;  // key not found
}
int mid = (low + high) / 2;

What if length > $2^{30}$?

... then we could have:  
low = $2^{30}$ = 0x40000000  
high = $2^{30}+1 = 0x40000001$  
low + high = $2^{31}+1 = 0x80000001$

Oops, in two’s complement, this is a negative number!

(low + high) / 2 = 0xC0000000  
= -3221225472

int midVal = a[mid];

Crashes because mid < 0
How can we fix the bug?

```c
int mid = (low + high) / 2;

int mid = low + ((high - low) / 2);
```

(There are other ways, but I think this is the simplest to understand)
This was an actual bug in Java

java.util.Arrays.binarySearch

This bug went unnoticed for years.
See: http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

Understanding binary number representations is important!
Check your textbook:
Don’t use the international edition!
The homework problems are different.
Today

• Questions on Hw 2 or Lab 2?

• Procedure calls
Procedure Call Example

**Caller**

```c
int z = sum(1, 2);
```

**Callee**

```c
int sum(int x, int y) {
    return x + y;
}
```
**Procedure Call Example**  
(IA32/Linux)

**Caller**

```
int z = sum(1, 2);
```

**Caller in assembly**

```
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

*note: these instruction addresses are completely made up for this example*
Procedure Call Example
(IA32/Linux)

**Caller**

```
int z = sum(1, 2);
```

**Caller in assembly**

```
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

**The Stack**

*note: these instruction addresses are completely made up for this example*
Procedure Call Example
(IA32/Linux)

Caller

```c
int z = sum(1, 2);
```

**Caller in assembly**

```
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

*note: these instruction addresses are completely made up for this example*
Procedure Call Example
(IA32/Linux)

**Caller**

```c
int z = sum(1, 2);
```

**Caller in assembly**

```assembly
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

**The Stack**

```
2
1
0x8013
```

*note: these instruction addresses are completely made up for this example*
Procedure Call Example  
(IA32/Linux)

Callee

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

Callee in assembly (simple version)

```
movl 8(%esp), %edi
movl 4(%esp), %eax
addl %edi, %eax
ret
```

The Stack

```
<table>
<thead>
<tr>
<th>8(%esp)</th>
<th>4(%esp)</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0x8013</td>
</tr>
</tbody>
</table>
```

Registers

```
%edi  2
```
Procedure Call Example
(IA32/Linux)

Callee

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

Callee in assembly (simple version)

```
    movl 8(%esp), %edi
    movl 4(%esp), %eax
    addl %edi, %eax
    ret
```

The Stack

```
<table>
<thead>
<tr>
<th>%esp</th>
<th>0x8013</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x8013</td>
</tr>
<tr>
<td>%eax</td>
<td>2</td>
</tr>
<tr>
<td>%edi</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Registers

```
| %eax | 1      |
| %edi | 2      |
```
Callee

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

Callee in assembly (simple version)

```
movl 8(%esp), %edi
movl 4(%esp), %eax
addl %edi, %eax
ret
```

%eax has the return value!

The Stack

```
8(%esp) ─→ 2
4(%esp) ─→ 1
%esp ─→ 0x8013
```

Registers

```
%eax 3
%edi 2
```
Procedure Call Example
(IA32/Linux)

Callee

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

Callee in assembly (simple version)

```assembly
movl 8(%esp), %edi
movl 4(%esp), %eax
addl %edi, %eax
ret
```

%eax has the return value!

The Stack

```
  2
  1
  x
  y
```

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>3</td>
</tr>
<tr>
<td>%edi</td>
<td>2</td>
</tr>
<tr>
<td>%eip</td>
<td>0x8013</td>
</tr>
</tbody>
</table>

%esp has the return value!
Procedure Call Example
(IA32/Linux)

**Caller**

```c
int z = sum(1, 2);
```

**Caller in assembly**

```assembly
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

---

**The Stack**

The stack contents are shown with the ESP pointer pointing to the top of the stack.

---

** Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>3</td>
</tr>
<tr>
<td>%edi</td>
<td>2</td>
</tr>
<tr>
<td>%eip</td>
<td>0x8013</td>
</tr>
</tbody>
</table>

*note: these instruction addresses are completely made up for this example*
Procedure Call Example
(IA32/Linux)

**Caller**

```c
int z = sum(1, 2);
```

**Caller in assembly**

```
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

**The Stack**

```
  :  :  
  2  
  1  
```

**Registers**

```
%eax  3
%edi  2
%eip  0x8013
```

*note: these instruction addresses are completely made up for this example*
Procedure Call Example
(IA32/Linux)

**Caller**

```c
int z = sum(1, 2);
```

**Caller in assembly**

```
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum

→ 0x8013  addl $8, %esp
```

**Problem:**
- What if **Caller** used `%edi` before making the call?

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>3</td>
</tr>
<tr>
<td>%edi</td>
<td>2</td>
</tr>
<tr>
<td>%eip</td>
<td>0x8013</td>
</tr>
</tbody>
</table>

*note: these instruction addresses are completely made up for this example*
Procedure Call Example
(IA32/Linux)

**Caller**

```c
int d = 5;
int z = sum(1, 2);
```

**Caller in assembly**

```
0x7fff movl $5, %edi
0x8001 pushl $2
0x8005 pushl $1
0x8009 call sum
0x8013 addl $8, %esp
```

**Problem:**
- What if **Caller** used `%edi` before making the call?

**sum() overwrote `%edi`!**
Need to save ...

**Registers**

- `%eax`:
- `%edi`:
  - 2
- `%eip`:
  - 0x8013

*note: these instruction addresses are completely made up for this example*
Saving Registers

• Some are **caller save**
  - IA32: `%eax, %edx, %ecx`
  - These are very commonly used
    (caller should expect they will be clobbered)

• Some are **callee save**
  - IA32: `%ebx, %edi, %esi`
  - These are less commonly used

*from prior example*
Procedure Call Example
(IA32/Linux)

Callee

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

Callee in assembly (better version)

```
pushl %ebp
movl %esp, %ebp
pushl %edi
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack

```
%ebp →
:  :
:  :
2
1
0x8013
%esp →
x
y
```
Procedure Call Example
(IA32/Linux)

Callee

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

Callee in assembly (better version)

```
setup
pushl %ebp
movl %esp, %ebp
pushl %edi

body
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax

cleanup
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack

```
%ebp →
  ...
  2
  1
  0x8013
old %ebp
%esp →
  ...
  x
  y
```
Procedure Call Example
(IA32/Linux)

Callee

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

Callee in assembly (better version)

```
setup

pushl %ebp
movl %esp, %ebp
pushl %edi

body

movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax

cleanup

movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack

```
old %ebp
0x8013
1
2
...
...

%esp
%ebp

x
y
```
Procedure Call Example

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

Callee in assembly (better version)

<table>
<thead>
<tr>
<th>setup</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pushl %ebp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl %esp, %ebp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pushl %edi</td>
<td></td>
</tr>
<tr>
<td>body</td>
<td>movl 12(%ebp), %edi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl 8(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addl %edi, %eax</td>
<td></td>
</tr>
<tr>
<td>cleanup</td>
<td>movl (%esp), %edi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl %ebp, %esp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>popl %ebp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

The Stack

- : :
- 2
- 1
- 0x8013
- old %ebp
- old %edi
- saved %edi

Callee

%esp

x

y
Procedure Call Example
(IA32/Linux)

Callee

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

Callee in assembly (better version)

```
setup
pushl %ebp
movl %esp, %ebp
pushl %edi

body
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax

cleanup
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack

Key: %ebp is fixed for the entire function
Procedure Call Example
(IA32/Linux)

Callee

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

The Stack

![Stack diagram]

Callee in assembly (better version)

**setup**

- `pushl %ebp`
- `movl %esp, %ebp`
- `pushl %edi`

**body**

- `movl 12(%ebp), %edi`
- `movl 8(%ebp), %eax`
- `addl %edi, %eax`

**cleanup**

- `movl (%esp), %edi`
- `movl %ebp, %esp`
- `popl %ebp`
- `ret`

- Restoring `%edi`
Procedure Call Example
(IA32/Linux)

Callee

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

Callee in assembly (better version)

setup

pushl %ebp
movl %esp, %ebp
pushl %edi

body

movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax

cleanup

movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret

The Stack

%ebp →

%esp →

0x8013

old %ebp

old %edi

2

1

y

x

restoring %ebp
Why use a frame pointer? (%ebp)

Callee

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

To make debugging easier
- `%esp` may move
- `%ebp` is fixed

Your compiler emits a symbol map
- `y → 12(%ebp)`
- `x → 8(%ebp)`

gdb uses this map when you write
- `print x`
Aside: how does gdb’s “backtrace” work?

Follow return addresses!
- use old %ebp to find prior frame

Pseudocode:
while (pc is not in “main”) {
  pc = 4(%ebp)
  %ebp = (%ebp)
}

The Stack

Pseudocode:
How is x86-64 different?

• Pass the first six arguments in registers
  - In this order: %rdi, %rsi, %rdx, %rcx, %r8, %r9

• New register save convention
  - Callee save: %rbx, %rbp, %r12, %r13, %r14, %r15
  - Others are caller save

• By default, gcc omits the frame pointer
  - It has to emit more complex debug info
    (e.g., the location of argument x relative to %esp can change)
Procedure Call Example
(x86-64/Linux)

**Caller**

```c
int z = sum(1, 2);
```

**Callee**

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

**Caller in assembly**

```assembly
movl $1, %edi
movl $2, %esi
call sum
```

**Callee in assembly**

```assembly
addl %esi, %edi
movl %edi, %eax
ret
```

**Tip:** you can force gcc to emit code with a frame pointer using `gcc -fno-omit-frame-pointer`

*edi* not *rdi* because *int* is 32-bits

*x86-64 with gcc* does not use a frame pointer

```c
int z = sum(1, 2);
```