Question 6.4–1: (Solution, p 4) In class, we examined an alternative to using a stack for supporting subroutines, where each subroutine would have its own static memory locations for remembering data. For example, consider the following `square()` function.

```
int square(int n) {
    return n * n;
}
```

This would be translated as follows into x86 assembly.

```
.section .data
sq_n:     .long 0                 # for passing the parameter value into square
sq_ret:   .long 0                 # remembers return address within caller of square
.section .text
square:   movl (sq_n), %eax
          imull %eax
          jmp *(sq_ret)
```

We saw that this scheme was wasteful of space and was not amenable to recursion.

a. Explain why stack allocation uses memory more efficiently than this described static allocation alternative.

b. Explain why recursion is difficult with static allocation.

Question 7.1–1: (Solution, p 4)

The x86 assembly code at right is a straightforward translation of the following C fragment.

```
do {
    ecx += 2 * edi;
    ecx -= 2 * esi;
    esi++;
} while(esi < edi);
```

```
up:   movl %edi, %eax         # ecx += 2 * edi;
      imull $2
      addl %eax, %ecx

      movl %esi, %eax         # ecx -= 2 * esi;
      imull $2
      subl %eax, %ecx

      incl %esi               # esi++;

      cmpl %edi, %esi         # if(esi < edi) goto up;
      jl up
```

Identify which of the following optimization techniques each of the following most represents.

A. peephole optimization

B. common subexpression elimination

C. strength reduction

a.

```
      movl %edi, %eax
      imull $2
      movl %eax, ebx
up:   addl %ebx, %ecx
      movl %esi, %eax
      imull $2
      subl %eax, %ecx
      incl %esi
      cmpl %edi, %esi
      jl up
```

b.

```
      movl %esi, %ebx
      addl %ebx, %ebx
up:   movl %edi, %eax
      imull $2
      addl %eax, %ecx
      subl %ebx, %ecx
      incl %esi
      addl $2, %ebx
      cmpl %edi, %esi
      jl up
```

c.

```
up:   movl %edi, %eax
      addl %eax, %eax
      addl %eax, %ecx
      movl %esi, %eax
      addl %eax, %eax
      subl %eax, %ecx
      incl %esi
      cmpl %edi, %esi
      jl up
```

Question 7.1–2: (Solution, p 4)

Consider the following C code with its Intel translation at right. The assembly translation uses `ecx` for `i`, `ebx` for `n`, and `esi` for `j`.

```
for(int i = 0; i < n; i++)
    j += 2 * i + 1;
```

```
        xorl %ecx, %ecx
again:  cmpl %ebx, %ecx
        jge done

        movl $2, %eax
        mull %ecx
        addl $1, %eax
        addl %eax, %esi

        incl %ecx
        jmp again
done:
```

For each of the following, select which of the following optimization techniques is being applied.

A. peephole optimization

B. common subexpression elimination

C. strength reduction

D. loop unrolling

a.

```
        xorl %ecx, %ecx
again:  cmpl %ebx, %ecx
        jge done

        movl $2, %eax
        mull %ecx
        addl $1, %eax
        addl %eax, %esi

        incl %ecx
        cmpl %ebx, %ecx
        jge done

        movl $2, %eax
        mull %ecx
        addl $1, %eax
        addl %eax, %esi

        incl %ecx
        jmp again
done:
```

b.

```
        xorl %ecx, %ecx
again:  cmpl %ebx, %ecx
        jge done

        leal 1(%esi, %eax, 2), %esi

        incl %ecx
        jmp again
done:
```

c.

```
        xorl %ecx, %ecx
        movl $1, %edx
again:  cmpl %ebx, %ecx
        jge done

        addl %edx, %esi

        incl %ecx
        addl $2, %edx
        jmp again
done:
```

Question 7.1–3: (Solution, p 4)

The C code below translates to the assembly language at right. The assembly code uses `ecx` for holding `i`, `ebx` for holding `n`, and `edi` for holding `a`.

```
for(i = 0; i < n; i++) {
    a[i] = 2 * n - 2 * i + 1;
}
```

Rewrite the assembly code below to illustrate the following two optimization techniques.

a. Common subexpression elimination

b. Strength reduction

```
        xorl %ecx, %ecx
        cmpl %ebx, %ecx
        jge done
again:  movl %ebx, %eax
        shll $1, %eax
        movl %ecx, %edx
        shll $1, %edx
        subl %edx, %eax
        incl %eax
        movl %eax, (%edi, %ecx, 4)

        incl %ecx
        cmpl %ebx, %ecx
        jl again
done:
```

Question 7.1–4: (Solution, p 5)

The C code below translates to the assembly language at right. The assembly code uses `ecx` for holding `i`, `ebx` for holding `n`, and `edi` for holding `a`.

```
for(i = 0; i < n; i++) {
    a[i] = 23 * i;
}
```

Rewrite the assembly code at right to illustrate the optimization technique of strength reduction.

```
        xorl %ecx, %ecx
        cmpl %ebx, %ecx
        jge done
again:  movl %ecx, %eax
        imull $23
        movl %eax, (%edi, %ecx, 4)

        incl %ecx
        cmpl %ebx, %ecx
        jl again
done:
```

Question 7.1–5: (Solution, p 5) Under what conditions can a compiler identify a recursive function as being tail-recursive, and hence eligible for having the recursive call optimized out?

Solution 6.4–1: (Question, p 1)

a. With stack allocation, only those subroutines currently being executed require memory. With static allocation, all subroutines, whether being executed or not, require memory for their data. As a result, much statically allocated space lies unused much of the time.

b. The problem arises when a subroutine wants to remember data through a recursive call. If the subroutine stores the needed data at a fixed location, then the recursive call, which itself will want to remember data through its own recursive call, will place its data at this same location. This will destroy the information saved there by the first call to the subroutine.

Solution 7.1–1: (Question, p 1)

a. B. common subexpression elimination

b. C. strength reduction

c. A. peephole optimization

Solution 7.1–2: (Question, p 2)

a. D. loop unrolling

b. A. peephole optimization

c. C. strength reduction

Solution 7.1–3: (Question, p 3)

a. Common subexpression elimination

```
        movl %ebx, %esi
        shll $1, %esi
        incl %esi
        xorl %ecx, %ecx
        cmpl %ebx, %ecx
        jge done
again:  movl %esi, %eax
        movl %ecx, %edx
        shll $1, %edx
        subl %edx, %eax
        movl %eax, (%edi, %ecx, 4)

        incl %ecx
        cmpl %ebx, %ecx
        jl again
done:
```

b. Strength reduction

```
        movl %ebx, %esi
        shll $1, %esi
        incl %esi
        xorl %ecx, %ecx
        cmpl %ebx, %ecx
        jge done
again:  movl %esi, (%edi, %ecx, 4)
        subl $2, %esi

        incl %ecx
        cmpl %ebx, %ecx
        jl again
done:
```

Solution 7.1–4: (Question, p 3)

```
        xorl %ecx, %ecx
        cmpl %ebx, %ecx
        jge done
        xorl %eax, %eax
again:  movl %ecx, %eax
        movl %eax, (%edi, %ecx, 4)

        addl $23, %eax
        incl %ecx
        cmpl %ebx, %ecx
        jl again
done:
```

Solution 7.1–5: (Question, p 3) The recursive call must be the last thing done before finishing the function. If the function is to return a value, the return value must be the same as the value returned by the recursive call.