Timing Part 1:
mergeSort() took 0.04 seconds, insertionSort() took 37.22 seconds

Timing Part 2:
mergeSort() took 0.02 seconds, insertionSort() took 30.86 seconds

Human vs. Compiler Optimization
A good unoptimized algorithm (merge sort) is still faster than a bad algorithm (insertion sort) optimized with -o2

Parts of an executable

a. numNumbers

objdump -d -s -j .data assign1-0

```
0000000000602064 <numNumbers>:
  602064:        00 00 02 00
```

b. String constant

objdump -d -s -j .rodata assign1-0

```
400da8:       25 64 20 20 00 00 00 00 48 6f 77 20 64 6f 20 79     %d  ....How do y
400db8:       6f 75 20 77 61 6e 74 20 74 6f 20 73 6f 72 74 20     ou want to sort
400dc8:       25 64 20 6e 75 6d 62 65 72 73 3f 0a 28 31 29 20     %d numbers?.(1)
400dd8:       49 6e 73 65 72 74 69 6f 6e 20 73 6f 72 74 0a 28     Insertion sort.(
400de8:       32 29 20 4d 65 72 67 65 20 73 6f 72 74 0a 59 6f     2) Merge sort.Yo
400df8:       75 72 20 63 68 6f 69 63 65 20 28 31 20 6f 72 20     ur choice (1 or
400e08:       32 29 3f 20 00 00 00 00                             2)? ....
```

c. print() code

objdump -d -j .text assign1-0

```
0000000000400a10 <print>:
  400a10:       55                      push   %rbp
  400a11:       48 89 e5                mov    %rsp,%rbp
  400a14:       53                      push   %rbx
  400a15:       48 83 ec 08             sub    $0x8,%rsp
  400a19:       e8 e2 fc ff ff          callq  400700 <mcount@plt>
  400a1e:       48 89 fb                mov    %rdi,%rbx
  400a21:       48 85 ff                test   %rdi,%rdi
  400a24:       74 21                   je     400a47 <print+0x37>
  400a26:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  400a2d:       00 00 00
  400a30:       8b 33                   mov    (%rbx),%esi
  400a32:       bf a8 0d 40 00          mov    $0x400da8,%edi
  400a37:       31 c0                   xor    %eax,%eax
  400a39:       e8 72 fc ff ff          callq  4006b0 <printf@plt>
  400a3e:       48 8b 5b 08             mov    0x8(%rbx),%rbx
  400a42:       48 85 db                test   %rbx,%rbx
  400a45:       75 e9                   jne    400a30 <print+0x20>
  400a47:       48 83 c4 08             add    $0x8,%rsp
  400a4b:       5b                      pop    %rbx
  400a4c:       5d                      pop    %rbp
  400a4d:       c3                      retq
  400a4e:       66 90                   xchg   %ax,%ax
```

d. The choice variable is on the stack since no memory can be allocated until input is given

Compiler optimizations

-o0

```
mov    0x8(%rax),%rax
mov    %rax,-0x30(%rbp)
cmpq   $0x0,-0x30(%rbp)
jne    400cba <insertionSort+0x54>
```

-o2

```
mov    0x8(%rdi),%r9
test   %r9,%r9
jne    400c25 <insertionSort+0x25>
```

Both sets of instructions want to see if the dereferenced variable is equal to 0 and jump if true. “test %r9, %r9” is an optimization via reduction in strength, as it uses bitwise & instead of using up memory on the stack with %rbp.

-o0

```
movq   $0x0,-0x8(%rbp)

movq   $0x0,-0x10(%rbp)
```

-o2

```
xor    %r10d,%r10d
xor    %ebx,%ebx
```

Similarly, “xor” sets both the %r10d and %ebx registers to 0, which is more efficient than moving 0’s onto the stack at -0x8(%rbp) and -0x1(%rbp)