CS107 Lecture 18: Two-Minute Etudes

Each function below is shown with one or two lines of C replaced by blanks, alongside the complete x86-64 **`gcc`**-generated assembly (including the instructions for the missing lines).

This time the blanks are mostly conditions and loop headers. Read each **`cmp`** or **`test`** with the jump right after it to decide if you're branching or not. Note: **`gcc`** often tests the **opposite** of the stated condition, so it can jump past the code that runs when your condition is true.

Problem: Reverse Compilation, Take III

a. Each blank is a straightforward test that dispatches toward one of three different **`ret`** instructions. And here, **`testq %rsi, %rsi`** followed by **`js`** jumps when **`y`** is negative.

C Code:

```
long
trio(long x, long y) {
  long result;
  if (__________) {
    result = x - y;
  } else if (__________) {
    result = y * 4;
  } else {
    result = -x;
  }
  return result;
}
```

x86:

```
trio: cmpq    $10, %rdi
      jle     .L6
      movq    %rdi, %rax
      subq    %rsi, %rax
      ret
.L6:  testq   %rsi, %rsi
      js      .L8
      leaq    0(,%rsi,4), %rax
      ret
.L8:  movq    %rdi, %rax
      negq    %rax
      ret
```

b. Similarly, **`testb %al, %al`** followed by **`je`** jumps when **`%al`** is zero.

C Code:

```
int crawl(const char *s,
          char ch) {
  int n = 0;
  while (_____________) {
    if (_____________) {
      n++;
    }
    s++;
  }
  return n;
}
```

x86:

```
crawl: movl    $0, %edx
       jmp     .L10
.L11:  addq    $1, %rdi
.L10:  movzbl  (%rdi), %eax
       testb   %al, %al
       je      .L13
       cmpb    %sil, %al
       jne     .L11
       addl    $1, %edx
       jmp     .L11
.L13:  movl    %edx, %eax
       ret
```

c. Here comes Euclid's algorithm, which manages to efficiently compute the largest integer that evenly divides both numbers. Remember that **`cqto`** sign-extends the 64-bit value in **`%rax`** through **`%rdx`**, producing the 128-bit dividend in **`%rdx:%rax`** that **`idivq`** expects. **`idivq %rsi`** then divides that dividend by **`%rsi`** and leaves the quotient in **`%rax`** and the remainder in **`%rdx`**.

C Code:

```
long
euclid(long a, long b) {
  while (_____________) {
    long r = ____________;
    a = b;
    b = r;
  }
  return a;
}
```

x86:

```
euclid: movq    %rdi, %rax
        jmp     .L2
.L3:    cqto
        idivq   %rsi
        movq    %rsi, %rax
        movq    %rdx, %rsi
.L2:    testq   %rsi, %rsi
        jne     .L3
        ret
```

d. Here, the loop searches from one end of the array toward the other. **`testq`** and **`js`** behave just as they did for part a.

C Code:

```
long
sift(int arr[], long n,
     int k) {
  for (_________________) {
    if (_____________) {
      return i;
    }
  }
  return -1;
}
```

x86:

```
sift: leaq    -1(%rsi), %rax
      jmp     .L18
.L22: subq    $1, %rax
.L18: testq   %rax, %rax
      js      .L21
      cmpl    %edx, (%rdi,%rax,4)
      jne     .L22
      ret
.L21: movq    $-1, %rax
      ret
```

Answers:

a. `x > 10 and y >= 0`

b. `*s != '\0' and *s == ch`

c. `b != 0 and a % b`

d. `long i = n - 1; i >= 0; i-- and arr[i] == k`