Big Oh Answers

1. (Medium) What is the Big-Oh runtime for the following recursive function in terms of input n?

```
1  int foo(int n)
2  {
3      if (n <= 1)
4          return n;
5      int ans = 0, i, j;
6      for (i = 0; i < n; i++)
7      {
8          for (j = 0; j < n; j++)
9          {
10             ans++;
11         }
12     }
13     int n1 = (n - 1) / 2;
14     int n2 = (n - 1) - n1;
15     ans += foo(n1);
16     ans += foo(n2);
17     return ans;
18 }
```

We must first find the recurrence relation. In our case we should find how many times we call the function foo in our function and how the input parameter is changed. The function foo(n) calls foo(n1) and foo(n2). Both n1 and n2 are roughly n/2, so we can says we call foo(n/2) twice.

$$T(n) = 2T(n/2) + ...$$

The lines from 6 to 12 behave in a n^2 manner. Thus the recurrence is

$$T(n) = 2T(n/2) + n^2$$

when not in the base case.

The base case come from lines 3 and 4. It gives us $T(1) = 1$ or some small constant. We now need to solve the following,

$$\begin{gathered} T(n) = 2T(n/2) + n^2 \text{ (when } n > 1) \\ T(1) = 1 \end{gathered}$$

Now for back substitution

$$\begin{aligned}
T(n) &= 2T(n/2) + n^2 \\
T(n/2) &= 2T(n/2/2) + (n/2)^2 \\
&= 2T(n/4) + (n^2/2^2) \\
T(n) &= 2\big(2T(n/4) + (n^2/2^2)\big) + n^2 \\
&= 4T(n/4) + (n^2/2) + n^2 \\
T(n/4) &= 2T(n/4/2) + (n/4)^2 \\
&= 2T(n/8) + (n^2/4^2) \\
T(n) &= 4\big(2T(n/8) + (n^2/4^2)\big) + (n^2/2) + n^2 \\
&= 8T(n/8) + (n^2/4) + (n^2/2) + n^2 \\
&= 2^kT(n/2^k) + \sum_{i=0}^{k-1} n^2/2^i \\
&= 2^kT(n/2^k) + n^2 \sum_{i=0}^{k-1} (1/2)^i \\
&= 2^kT(n/2^k) + n^2 \frac{(1/2)^k - (1/2)^0}{(1/2) - 1}
\end{aligned}$$

Let $T(n/2^k) = T(1)$. Thus $n/2^k = 1$, $n = 2^k$, and $k = \log_2(n)$

$$\begin{aligned}
T(n) &= 2^k T(n/2^k) + n^2 \frac{(1/2)^k - (1/2)^0}{(1/2) - 1} \\
&= nT(1) + n^2 \frac{(1/n) - 1}{-(1/2)} \\
&= n(1) + 2n^2 - 2n \\
&= 2n^2 - n \\
&\in O(n^2)
\end{aligned}$$

2. (Medium-Hard) What is the Big-Oh runtime for the following recursive function in terms of input n?

```
1  int foo(int n, int * arr)
2  {
3      if (n == 1)
4          return arr[0];
5      if (n == 0)
6          return 0;
7      int res = 0, i;
8      for (i = 0; i < n; i++)
9          res += arr[i];
10     res += foo(n/2, arr);
11     res += foo(n/2, arr + (n / 4));
12     res += foo(n/2, arr + (n / 2));
13     return res;
14 }
```

We need to find the recurrence relation in terms of the input parameters. It appears that n is the value that affects the runtime. The values in array itself does not have an effect. foo calls itself 3 times. Each time the value n is halved. Thus we have the following basic recurrence relation.

$$T(n) = 3T(n/2) + ...$$

Lines 8 and 9 force the function to perform a linear number of operations at each recursive function call

$$T(n) = 3T(n/2) + n$$

The base case line 3 is when the function input parameter, n, is 1. Thus we need to solve the following recurrence relation.

$$\begin{gathered}
T(n) = 3T(n/2) + n \text{ (when } n > 1) \\
T(1) = 1
\end{gathered}$$

The math is below

$$\begin{aligned}
T(n) &= 3T(n/2) + n \\
T(n/2) &= 3T(n/2/2) + n/2 \\
&= 3T(n/4) + n/2 \\
T(n) &= 3(3T(n/4) + n/2) + n \\
&= 9T(n/4) + (3/2)n + n \\
T(n/4) &= 3T(n/4/2) + n/4 \\
&= 3T(n/8) + n/4 \\
T(n) &= 9(3T(n/8) + n/4) + (3/2)n + n \\
&= 27T(n/8) + (9/4)n + (3/2)n + n \\
T(n/8) &= 3T(n/8/2) + n/8 \\
&= 3T(n/16) + n/8 \\
T(n) &= 27(3T(n/16) + n/8) + (9/4)n + (3/2)n + n \\
&= 81T(n/16) + (27/8)n + (9/4)n + (3/2)n + n \\
&= (3^k)T(n/(2^k)) + \sum_{i=0}^{k-1} (3/2)^i n
\end{aligned}$$

$$\begin{aligned}
T(n) &= (3^k)T(n/(2^k)) + \sum_{i=0}^{k-1}(3/2)^i n \\
&= (3^k)T(n/(2^k)) + n\sum_{i=0}^{k-1}(3/2)^i \\
&= (3^k)T(n/(2^k)) + n\frac{(3/2)^k - (3/2)^0}{3/2-1}
\end{aligned}$$

To satisfy our base case we will find that $n/2^k = 1$, $n = 2^k$, and $\log_2(n) = k$. **[Note:** $a^{\mathbf{log}_b(c)} = c^{\mathbf{log}_b(a)}$**]**

$$\begin{aligned}
T(n) &= (3^{\log_2(n)})T(1) + n\frac{(3/2)^{\log_2(n)} - (3/2)^0}{3/2-1} \\
&= (n^{\log_2(3)})1 + n\frac{n^{\log_2(3/2)} - 1}{1/2} \\
&= (n^{\log_2(3)}) + n^1 2(n^{\log_2(3)-\log_2(2)} - 1) \\
&= (n^{\log_2(3)}) + 2(n^{1+\log_2(3)-1} - n^1) \\
&= (n^{\log_2(3)}) + 2(n^{\log_2(3)} - n) \\
&= 3(n^{\log_2(3)}) - 2n \\
&\in O(n^{\log_2(3)})
\end{aligned}$$

3. (Medium-Hard) What is the Big-Oh runtime for the following recursive function in terms of input n?

```
1  int foo(int n, int * arr)
2  {
3      if (n == 1)
4          return arr[0];
5      if (n == 0)
6          return 0;
7      int res = 0, i;
8      res += arr[0];
9      res += foo(n/2, arr);
10     res += foo(n/2, arr + (n / 4));
11     res += foo(n/2, arr + (n / 2));
12     return res;
13 }
```

We need to find the recurrence relation in terms of the input parameters. It appears that n is the value that affects the runtime. The values in array itself does not have an effect. foo calls itself 3 times. Each time the value n is halved. Thus we have the following basic recurrence relation.

$$T(n) = 3T(n/2) + ...$$

There is a constant number of operations for each foo call, so we can use 1 to represent the work done on each foo function in addition to it's recursive statements.

$$T(n) = 3T(n/2) + 1$$

The base case line 3 is when the function input parameter, n, is 1. Thus we need to solve the following recurrence relation.

$$\begin{gathered}
T(n) = 3T(n/2) + 1 \text{ (when } n > 1) \\
T(1) = 1
\end{gathered}$$

The math is below

$$\begin{aligned}
T(n) &= 3T(n/2) + 1 \\
T(n/2) &= 3T(n/2/2) + 1 \\
&= 3T(n/4) + 1 \\
T(n) &= 3(3T(n/4) + 1) + 1 \\
&= 9T(n/4) + 3 + 1 \\
T(n/4) &= 3T(n/4/2) + 1 \\
&= 3T(n/8) + 1 \\
T(n) &= 9(3T(n/8) + 1) + 3 + 1 \\
&= 27T(n/8) + 9 + 3 + 1 \\
T(n/8) &= 3T(n/8/2) + 1 \\
&= 3T(n/16) + 1 \\
T(n) &= 27(3T(n/16) + 1) + 9 + 3 + 1 \\
&= 81T(n/16) + 27 + 9 + 3 + 1 \\
&= (3^k)T(n/(2^k)) + \sum_{i=0}^{k-1} 3^i \\
&= (3^k)T(n/(2^k)) + \frac{3^k - 3^0}{3 - 1}
\end{aligned}$$

To satisfy our base case we will find that $n/2^k = 1$, $n = 2^k$, and $\log_2(n) = k$. **[Note:** $a^{\log_b(c)} = c^{\log_b(a)}$**]**

$$\begin{aligned}
T(n) &= (3^{\log_2(n)})T(1) + \frac{3^{\log_2(n)} - 3^0}{3 - 1} \\
&= (n^{\log_2(3)})1 + \frac{n^{\log_2(3)} - 1}{2} \\
&= (n^{\log_2(3)}) + (.5)(n^{\log_2(3)} - 1) \\
&= ((3/2)n^{\log_2(3)}) - 1/2 \\
&\in O(n^{\log_2(3)})
\end{aligned}$$

4. What is the best (Medium-Easy), average (Challenger), and worst (Medium-Easy) case runtime for the following segment of code in terms of n, where n is assumed to be positive?

```
int n, i;
scanf("%d", &n);
while (n != 0) {
    n = rand() % n;
    if (n < 0)
        n = -n;
    for (i = 0; i < n; i++)
        sum++;
}
```

Best Case. We get n = 0 in the first random assignment. That means the for loop does no loop and the while loop will terminate immediately. there is a constant number of operations $O(1)$

Worst Case. We cannot get n to be n since the mod returns a value in the range of 0 to n - 1. At worst we find n = n - 1 in the every random assignment. That means the for loop will loop n - 1 times. This will cause us to loop n - 1, then n - 2, then n - 3, and so on. The sum of these values turns into a quadratic growth or $O(n^2)$.

Average Case. We will most-likely need to use a recurrence relation. To make the recurrent relation easier we will over count the number of operations and then adjust to get the final answer.

Assume that we will run n extra operations. We can say that every value in the range of 0 to n-1 can be chosen with an equal likely probability (1/n). We can say that the recurrence relation is the following

$$T(n) = n + (1/n)(T(0) + T(1) + T(2) + T(3) + ... + T(n - 1))$$

We can multiply everything by n to get

$$nT(n) = n^2 + (T(0) + T(1) + T(2) + T(3) + ... + T(n-1))$$

By plugging in n-1 we get the following

$$(n-1)T(n-1) = (n-1)^2 + (T(0) + T(1) + T(2) + T(3) + ... + T(n-2))$$

Take the differences of both side from the two equations above and we get

$$nT(n) - (n-1)T(n-1) = n^2 - (n-1)^2 + T(n-1)$$

Some math gets us

$$nT(n) = 2n - 1 + T(n-1) + (n-1)T(n-1)$$

Divide by n

$$T(n) = 2 + T(n-1) - 1/n$$

By backsub we find

$$T(n) = \sum_{i=1}^{k} 2 + T(n-k) - \sum_{i=1}^{k}(1/i)$$

Letting $n = k$ we get (by harmonic series)

$$T(n) \approx 2n + 0 - c\log(n)$$

Note that this is n operations more than the correct answer, so the actual answer is $T(n) \approx n - c\log(n) \in O(n)$.

5. (Medium-Hard) What is the big-Oh runtime for the following segment of code?

```c
int n, i;
scanf("%d", &n);
for (i = 0; i < n; i++)
{
    int tn = i + 1;
    while (0 == (tn & 1))
    {
        sum++;
        tn /= 2; // or tn >>= 1;
    }
}
```

The program itself will loop in the while loop (over each integer), until the number has a 1 as the least significant digit. It loops a number of times equal to the position of the lowest 1 bit in a binary number. You don't need to worry too much about the terminology. We can write out how many times a number contributes to sum

value	sum contribution
0	0
1	1
2	0
3	2
4	0
5	1
6	0
7	3
$\vdots$	$\vdots$

We have n/2 terms that are 0, we have n/4 terms that are 1. In general we have $n/(2^k)$ terms that are $k-1$.

The answer becomes (for $k = \log_2(n) - 1$)

$$\begin{aligned} T(n) &= \sum_{i=0}^{k} i(n/2^i) \\ &= n\sum_{i=0}^{k}(i/2^i) \end{aligned}$$

To make the problem “easier” treat k as infinity. We will have an infinite sum and the result will be an upper bound.

Let $S = 1/2 + 2/4 + 3/8 + ...$

$$\begin{aligned}
S &= 1/2 + 2/4 + 3/8 + 4/16 + ... \\
(1/2)S &= 1/4 + 2/8 + 3/16 + ... \\
S - (1/2)S &= (1/2) + (2/4 - 1/4) + (3/8 - 2/8) + (4/16 - 3/16) + ... \\
(1/2)S &= 1/2 + 1/4 + 1/8 + 1/16 + ... \\
(1/2)S &= 1 \\
S &= 2
\end{aligned}$$

Thus the number of operation across all sum increments must be less than or equal to $2n$.

Since we loop over n values the number of operations will be exactly $\Theta(n)$.

6. (Medium-Hard) What is the big-Oh runtime for the following segment of code?

```c
int n, i, j;
scanf("%d", &n);
for (i = 1; i < n; i++)
{
    for (j = 0; j < n; j += i)
    {
        sum++;
    }
}
```

The number of operations performed in the inner loop is n/i, because j is incremented by i until it passes n, which if $j \approx n$, then $j/i \approx n/i$, where j/i is the number of times that j is incremented.

Thus the answer is $\sum_{i=1}^{n} n/i$, which is n times a harmonic series. The harmonic series converges to $\log(n)$, so the growth $\in O(n\log(n))$