1 Asymptotics Introduction

Give the runtime of the following functions in Θ notation. Your answer should be as simple as possible with no unnecessary leading constants or lower order terms.

```
private void f1(int N) {
    for (int i = 1; i < N; i++) {
        for (int j = 1; j < i; j++) {
            System.out.println("hello tony");
        }
    }
}
Θ(___)
```

Solution: $\Theta(N^2)$

Explanation: The inner loop does up to i work each time, and the outer loop increments i each time. Summing over each loop, we get that $1+2+3+4+\ldots+N = \Theta(N^2)$.

```
private void f2(int N) {
    for (int i = 1; i < N; i *= 2) {
        for (int j = 1; j < i; j++) {
            System.out.println("hello hannah");
        }
    }
}
Θ(___)
```

Solution: $\Theta(N)$

Explanation: The inner loop does i work each time, and we double i each time until reaching N. $1+2+4+8+\ldots+N = \Theta(N)$

Here is a video walkthrough of both parts.

2 Finish the Runtimes

Below we see the standard nested for loop, but with missing pieces!

```
for (int i = 1; i < ______; i = ______) {
    for (int j = 1; j < ______; j = ______) {
        System.out.println("We will miss you next semester Akshit :(");
    }
}
```

For each part below, **some** of the blanks will be filled in, and a desired runtime will be given. Fill in the remaining blanks to achieve the desired runtime! There may be more than one correct answer.

Hint: You may find `Math.pow` helpful.

(a) Desired runtime: $\Theta(N^2)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = ______) {
        System.out.println("This is one is low key hard");
    }
}
```

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = j + 1) {
        System.out.println("This is one is low key hard");
    }
}
```

Explanation: Remember the arithmetic series $1+2+3+4+\ldots+N = \Theta(N^2)$. We get this series by incrementing j by 1 per inner loop.

(b) Desired runtime: $\Theta(log(N))$

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < ______; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

Any constant would work here, 2 was chosen arbitrarily.

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < 2; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

Explanation: The outer loop already runs $\log n$ times, since i doubles each time. This means the inner loop must do constant work (so any constant `j < k` would work).

(c) Desired runtime: $\Theta(2^N)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < ______; j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < Math.pow(2, i); j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

Explanation: Remember the geometric series $1 + 2 + 4 + ... + 2^N = \Theta(2^N)$. We notice that i increments by 1 each time, so in order to achieve this 2^N runtime, we must run the inner loop 2^i times per outer loop iteration.

(d) Desired runtime: $\Theta(N^3)$

```
for (int i = 1; i < ______; i = i * 2) {
    for (int j = 1; j < N * N; j = ______) {
        System.out.println("yikes");
    }
}
```

```
for (int i = 1; i < Math.pow(2, N); i = i * 2) {
    for (int j = 1; j < N * N; j = j + 1) {
        System.out.println("yikes");
    }
}
```

Explanation: One way to get N^3 runtime is to have the outer loop run N times, and the inner loop run N^2 times per outer loop iteration. To make the outer loop run N times, we need stop after multiplying `i = i * 2` N times, giving us the condition `i < Math.pow(2, N)`. To make the inner loop run N^2 times, we can simply increment by 1 each time.

3 Bit Operations

In the following questions, use bit manipulation operations to achieve the intended functionality and fill out the function details -

(a) Implement a function `isPalindrome` which checks if the binary representation of a given number is palindrome. The function returns true if and only if the binary representation of `num` is a palindrome.

For example, the function should return true for `isPalindrome(9)` since binary representation of 9 is `1001` which is a palindrome.

```
/**
* Returns true if binary representation of num is a palindrome
*/
public static boolean isPalindrome(int num) {
    // stores reverse of binary representation of num
    int reverse = 0;

    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    return num == reverse;
}
```

Solution:

```
/**
* Returns true if binary representation of num is a palindrome
*/
public static boolean isPalindrome(int num) {
    // stores reverse of binary representation of num
    int reverse = 0;

    // do till all bits of num are processed
    int k = num;
```

```
    while (k > 0)
    {
        // add rightmost bit to reverse
        reverse = (reverse << 1) | (k & 1);
        k = k >> 1;              // drop last bit
    }
    return num == reverse;
}
```

Explanation: The main idea is to reverse the bits of `num`; it is a palindrome if and only if it is equal to its reverse. To do this, we initialize `reverse` to all zeros. Inside the loop:

1. Shift `reverse` to "vacate" its last bit.
   `rrr << 1 -> rrr0`
2. Get the last bit of `k`.
   `kkkk & 0001 -> 000k`
3. `or` the numbers together to get the combined bits.
   `rrr0 | 000k -> rrrk`
4. Remove the bit of `k` we just used.

(b) Implement a function `swap` which for a given integer, swaps two bits at given positions. The function returns the resulting integer after bit swap operation.

For example, when the function is called with inputs `swap(31, 3, 7)`, it should reverse the 3rd and 7th bits from the right and return 91 since 31 (00011111) would become 91 (01011011).

```
/**
* Function to swap bits at position a and b (from right) in integer num
*/
public static int swap(int num, int a, int b) {
    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    _________________________________________________

    return num;
}
```

Solution:

```
/**
* Function to swap bits at position a and b (from right) in integer num
*/
public static int swap(int num, int a, int b) {
    int p = a-1;
    int q = b-1;

    int bit_a = (num >> p) & 1;
    int bit_b = (num >> q) & 1;

    if (bit_a != bit_b) {       // if the bits are different
        num ^= (1 << p);
        num ^= (1 << q);
    }
    return num;
}
```

Explanation: To get the `k`th bit from the right in a number, we can shift the number right by `k - 1` bits, then perform an with 1. For a visualization, suppose we are trying to get the third bit from the right for $b_4b_3b_2b_1$. First, we right shift by 2 to get $00b_4b_3$. $00b_4b_3$ & 0001 gives $000b_3$ as desired. This is the operation performed in line 8 and 9.

We only need to swap if the two bits are different. If the bits are different, this problem reduces to flipping the bits at position `a` and `b`. To flip a bit at position `k`, we simply `xor` it with 1 ($1 \oplus 1 = 0, 0 \oplus 1 = 1$). This corresponds to lines 12 and 13.

4 Bits Runtime

Determine the best and worst case runtime of `tricky`.

```
public void tricky(int n) {
    if (n > 0) {
        tricky(n & (n - 1));
    }
}
```

Best Case: Θ(   ), Worst Case: Θ(   )

Solution:
Best Case: $\Theta(1)$, Worst Case: $\Theta(logN)$

Explanation: The main idea is that this function zeros out a `1` in `n` each time. If `n` starts off as some power of 2, it only has one `1` and finishes in constant time. If `n` is all ones, it takes `log N` recursive calls to finish (there are `log N` bits in N).

There are two main cases for `n`. First, if `n` is odd, `n - 1` has a `0` in the last bit, so the last bit of `n` will be zeroed out. If `n` is even so its last bits are something like `10 ... 0`, then the last bits of `n - 1` will be `01 ... 1`. `and`-ing these together zeros out the first nonzero bit from the right.