1  Round Down

Given some power of two powerOfTwo and a positive number num, round num down to the nearest multiple of powerOfTwo. Assume powerOfTwo is greater than or equal to 1. You may use only bit operations and one subtraction/addition operation.

Examples:

1. roundDown(8, 53) -> 48
2. roundDown(16, 90) -> 80
3. roundDown(1, 90) -> 90

```java
public int roundDown(int powerOfTwo, int num) {
    return _____________________________________________________;
}
```

2  Heaps

a) (2.5 Points) i) (1 Point). Suppose we have the min-heap below (represented as an array) with distinct elements, where the values of A and B are unknown. Note that A and B aren’t necessarily integers.

{1, A, 3, 5, 9, 11, 13, 10, B}

What can we say about the relationships between the following elements? Put >, <, or ? if the answer is not known.

A  O >  O <  O ? 1

A  O >  O <  O ? 3

B  O >  O <  O ? 10

A  O >  O <  O ? B

ii) (1.5 Points). Note for both parts below, the values of A and B should not violate the min-heap properties. Put -inf or inf if there isn’t a lower or upper bound, respectively. If the bound for B depends on the value of A, or vice versa, you may put the variable in the bound, e.g. A < B.

Considering one removeMin call, put tight bounds on A and B such that:
• We perform the **maximum** number of swaps.

   _____ < A < _____

   _____ < B < _____

• We perform the **minimum** number of swaps.

   _____ < A < _____

   _____ < B < _____
3 Hashing Asymptotics

Suppose we set the hashCode and equals methods of the ArrayList class as follows.

```java
/* Returns true iff the lists have the same elements in the same ordering */
@override
public boolean equals(Object o) {
    if (o == null || o.getClass() != this.getClass() || o.size() != this.size()) {
        return false;
    }
    ArrayList<T> other = (ArrayList<T>) o;
    for (int i = 0; i < this.size(); i++) {
        if (other.get(i) != this.get(i)) {
            return false;
        }
    }
    return true;
}

/* Returns the sum of the hashCodes in the list. Assume the sum is a cached instance variable. */
@override
public int hashCode() {
    return sum;
}
```

(a) Give the best and worst case runtime of hashContents in Θ(.) notation as a function of N, where N is initial size of the list. Assume the length of set's underlying array is N and the set does not resize. Assume the hashCode of an Integer is itself. Admittedly, the ArrayList class does not have the method removeLast, but assume it does for this problem, and is implemented in amortized constant time. Finally, assume f accepts two integers, returns an unknown integer, and runs in constant time.

```java
static void hashContents(HashSet<ArrayList<Integer>> set, ArrayList<Integer> list) {
    if (list.size() <= 1) {
        return;
    }
    int last = list.removeLast();
    list.set(0, f(list.get(0), last));
    set.add(list);
    hashContents(set, list);
}
```

Best Case: Θ( ), Worst Case: Θ( )
(b) Continuing from the previous part, how can we define \( f \) to ensure the worst case runtime? How can we define \( f \) to ensure the best case runtime? There may be multiple possible answers.

1. Worst case:

```c
int f(int first, int last) {
    return __________________________;
}
```

2. Best case:

```c
int f(int first, int last) {
    return __________________________;
}
```
4 Boolean Confusion

Give the best and worst case runtime in $\Theta(.)$ notation as a function of $N$, where $N$ is $arr.length$. Your answer should be simple with no unnecessary leading constants or summations.

```java
void confusion(boolean[] arr) {
    boolean first = arr[0];
    int next;
    for (next = 1; arr[next] == first; next++) {
        if (next == arr.length - 1) {
            return;
        }
    }
    for (int i = 0; i < next; i++) {
        arr[i] = !arr[i];
    }
    confusion(arr);
}
```

Best Case: $\Theta( )$, Worst Case: $\Theta( )$

5 Gamma

Give the best and worst case runtime in $\Theta(.)$ notation as a function of $N$. Your answer should be simple with no unnecessary leading constants or summations. Assume $f(N)$ returns a random number between 1 and $N/2$, inclusive, and does so in constant time.

```java
static void gamma(int N) {
    if (N <= 10) {
        return;
    }
    for (int i = f(N); i < N; i += f(N)) {
        gamma(i);
    }
}
```

Best Case: $\Theta( )$, Worst Case: $\Theta( )$