1 Round Down

Here is a video walkthrough of the solutions.

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

Examples:

1. \( \text{roundDown}(8, 53) \rightarrow 48 \)
2. \( \text{roundDown}(16, 90) \rightarrow 80 \)
3. \( \text{roundDown}(1, 90) \rightarrow 90 \)

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

Solution:

```java
public int roundDown(int powerOfTwo, int num) {
    return ~(powerOfTwo - 1) & num;
}
```

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 □ > □ < □ ? 1

A □ > □ < □ ? 3

B □ > □ < □ ? 10

A □ > □ < □ ? B
Solution:

Here is a video walkthrough of the solutions.

\[ A \sqrt{ > \circ < \circ \circ \circ } \ 1 \]

\[ A \circ > \circ < \sqrt{ ? } \ 3 \]

\[ B \circ > \circ < \sqrt{ ? } \ 10 \]

\[ A \circ > \sqrt{ < \circ \circ } \ B \]

**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.
  \[ \ldots < A < \ldots \]
  \[ \ldots < B < \ldots \]

- We perform the **minimum** number of swaps.
  \[ \ldots < A < \ldots \]
  \[ \ldots < B < \ldots \]

**Solution:**

Here is a video walkthrough of the solutions.

- We perform the **maximum** number of swaps.
  \[ 1 < A < 3 \]
  \[ 10 < B < \text{inf} \]

- We perform the **minimum** number of swaps.
  \[ 3 < A < 5 \]
  \[ 5 < B < 11 \]
3 Hashing Asymptotics

Here is a video walkthrough of the solutions.

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 $\Theta(\cdot)$ 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 `ints`, returns an unknown `int`, 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: $\Theta(\cdot)$, Worst Case: $\Theta(\cdot)$

**Solution:**

Best Case: $\Theta(N)$, Worst Case: $\Theta(N^2)$
(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 ______________________;
}
```

**Solution:**

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

2. Best case:

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

**Solution:**

```c
int f(int first, int last) {
    return first + last + 1;
}
```

**Alternate solution:**

```c
int f(int first, int last) {
    return first + last - 1;
}
```
4  Boolean Confusion

Here is a video walkthrough of the solutions.

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: \( \Theta( ) \)

**Solution:**

Best Case: \( \Theta(N) \), Worst Case: \( \Theta(N^2) \)

5  Gamma

Here is a video walkthrough of the solutions.

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: \( \Theta( ) \)

**Solution:**

Best Case: \( \Theta(log(N)) \), Worst Case: \( \Theta(2^N) \)