CS 61B Spring 2025

Iterators, Asymptotics

Exam-Level 04: February 17, 2025

1 Iterator of Iterators

Implement an **IteratorOfIterators** which takes in a **List** of **Iterator**s of **Integers** as an argument. The first call to **next()** should return the first item from the first iterator in the list. The second call should return the first item from the second iterator in the list. If the list contained **n** iterators, the **n+1**th time that we call **next()**, we would return the second item of the first iterator in the list.

Note that if an iterator is empty in this process, we continue to the next iterator. Then, once all the iterators are empty, **hasNext** should return **false**. For example, if we had 3 **Iterators** A, B, and C such that A contained the values **[1, 3, 4, 5]**, B was empty, and C contained the values **[2]**, calls to **next()** for our **IteratorOfIterators** would return **[1, 2, 3, 4, 5]**.

```
public class IteratorOfIterators ____________________________________________ {
    private List<Iterator<Integer>> iterators;
    private int curr;
    public IteratorOfIterators(List<Iterator<Integer>> a) {
        iterators = new LinkedList<>();
        for (_____________________________________________) {
            if (_____________________________________________) {
                _____________________________________________;
            }
        }
        curr = 0;
    }
    @Override
    public boolean hasNext() {
        return _____________________________________________;
    }
    @Override
    public Integer next() {
        if (!hasNext()) { throw new NoSuchElementException(); }
        Iterator<Integer> currIterator = _____________________________________________;
        int result = _____________________________________________;
        if (_____________________________________________) {
            _____________________________________________;
            if (iterators.isEmpty()) {
                _____________________________________________;
            }
        } else {
            curr = _____________________________________________;
        }
        return result;
    }
}
```

Solution:

```
public class IteratorOfIterators implements Iterator<Integer> {
    private List<Iterator<Integer>> iterators;
    private int curr;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        iterators = new LinkedList<>();
        for (Iterator<Integer> iterator : a) {
            if (iterator.hasNext()) {
                iterators.add(iterator);
            }
        }
        curr = 0;
    }

    @Override
    public boolean hasNext() {
        return !iterators.isEmpty();
    }

    @Override
    public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Iterator<Integer> currIterator = iterators.get(curr);
        int result = iterators.get(curr).next();
        if (!currIterator.hasNext()) {
            iterators.remove(curr);
            if (curr >= iterators.size()) {
                curr = 0;
            }
        } else {
            curr = (curr + 1) % iterators.size();
        }
        return result;
    }
}
```

For this problem, we use the instance variable **`iterators`** to store all the iterators that still has elements. We use **`curr`** to indicate the next iterator to get the next element from. Therefore, in the constructor, we initialize **`iterators`** by iterating through the input list of iterators and adding the iterators that are not empty. We then initialize **`curr`** to 0. For the **`hasNext()`** method, we can test whether our list **`iterators`** is empty. For the **`next()`** method, we first check if there are any elements left to iterate through (throwing an error if we do not have). If there are, we get the current iterator and the next element from that iterator. If the iterator is empty, we remove it from the list of iterators. Otherwise, we increment **`curr`** to the next iterator. Notice that we do not increment **`curr`** if we remove an iterator from the list, as all the indices of the following iterators will shift by 1, and next iterator will take its place.

2 Asymptotics

(a) Say we have a function **findMax** that iterates through an unsorted int array one time and returns the maximum element found in that array. Give the tightest lower and upper bounds ($\Omega(\cdot)$ and $O(\cdot)$) of **findMax** in terms of **N**, the length of the array. Is it possible to define a $\Theta(\cdot)$ bound for **findMax**?

Solution: Lower bound: $\Omega(N)$, Upper bound: $O(N)$

Because the array is unsorted, we don't know where the max will be, so we have to iterate through the entire array to ensure that we find the true max. Therefore, we know that we can never go faster than linear time with respect to the length of the array. Since the function is both lower and upper bounded by N, we can say that the function is theta-bounded by N as well ($\Theta(N)$).

(b) Give the worst case and best case runtime in terms of M and N. Assume **ping** runs in $\Theta(1)$ and returns an **int**.

```
for (int i = N; i > 0; i--) {
    for (int j = 0; j <= M; j++) {
        if (ping(i, j) > 64) { break; }
    }
}
```

Solution: Worst: $\Theta(MN)$, Best: $\Theta(N)$ We repeat the outer loop N times, no matter what. For the inner loop, the amount of times we repeat it depends on the result of **ping**. In the best case, it returns **true** immediately, such that we'll only ever look at the inner loop once and then break the inner loop. In the worst case, **ping** is always **false** and we complete the inner loop **M** times for every value of **N** in the outer loop.

(c) Below we have a function that returns **true** if every **int** has a duplicate in the array, and **false** if there is any unique int in the array. Assume **sort(array)** is in $\Theta(N \log N)$ and returns **array** sorted.

```
public static boolean noUniques(int[] array) {
    array = sort(array);
    int N = array.length;
    for (int i = 0; i < N; i += 1) {
        boolean hasDuplicate = false;
        for (int j = 0; j < N; j += 1) {
            if (i != j && array[i] == array[j]) {
                hasDuplicate = true;
            }
        }
        if (!hasDuplicate) return false;
    }
    return true;
}
```

Give the worst case and best case runtime where $N =$ **array.length**.

Notice that we call `sort` at the beginning of the function, which we are told runs in $\Theta(N \log N)$.

First, we consider the best case. We notice that if `hasDuplicate` is false after the inner loop (i.e. `!hasDuplicate` has truth value `true`) we can exit the `for` loop early via the return statement on line 11. Thus, the best case is when we never set `hasDuplicate` to be `true` during the first time we run the inner loop. In this case, we can return after only looping through the array once, giving us $\Theta(N \log N + N) = \Theta(N \log N)$.

For the worst case, we notice that if `hasDuplicate` is always set to `true` by the inner loop, we never return on line 11. Thus, we consider the worst case where `hasDuplicate` is always set to `true` in every loop, forcing us to have to loop fully through both the inner and outer loop. One such input is an array of all the same integer! Since we have to fully loop through both loops, our worst-case runtime is $\Theta(N \log N + N^2) = \Theta(N^2)$.