1 Algorithm Analysis

Some Useful Formulas:

$$1 + 2 + 3 + 4 ... + N = \frac{N(N+1)}{2}$$

$$1 + 2 + 4 + 8 + ... + N = 2N - 1$$

$$2^0 + 2^1 + 2^2 + 2^3 + ... + 2^N = 2^{N+1} - 1$$

$$a + ar + ar^2 + ... + ar^{n-1} = \Sigma_{i=1}^{n} ar^{i-1} = a\frac{1-r^n}{1-r}$$

Give the worst-case order of growth of the runtime in $\Theta(\cdot)$ notation as a function of N. Your answer should be simple with no unnecessary leading constants or summations.

(a) $\Theta(\underline{N\quad})$

```
static int recursion(int N) {
    int x = 0;
    if (N < 1000) {
        for (int i = 0; i <= N * N * N; i++) {
            x++;
        }
        return x;
    }
    for (int i = 1; i <= N / 2; i++) {
        x++;
    }
    return x + 2 * recursion(N / 2);
}
```

The base case ($N < 1,000$) has a constant amount of work because N is relatively small (remember that we analyze the algorithm where N is very large). Hence, we have a recurrence like this:

$$T(N) = \begin{cases} c_0 & \text{if } N < 1,000 \\ T(\frac{N}{2}) + c_1 N & \text{otherwise} \end{cases}$$

Unrolling the recurrence gives us

$$\begin{aligned}
T(N) &= T\Big(\frac{N}{2}\Big) + c_1 N \\
&= T\Big(\frac{N}{4}\Big) + c_1\frac{N}{2} + c_1 N \\
&= T\Big(\frac{N}{8}\Big) + c_1\frac{N}{4} + c_1\frac{N}{2} + c_1 N \\
&= ... \\
&= T\Big(\frac{N}{2^k}\Big) + c_1 N\Big(\frac{1}{2^{k-1}} + ... + \frac{1}{4} + \frac{1}{2} + 1\Big) && \frac{N}{2^k} = 999; k = log_2\frac{N}{999} \\
&= T(999) + c_1 N\Big(1 \cdot \frac{1-\frac{1}{2}^k}{1-\frac{1}{2}}\Big) && \text{by finite geometric series} \\
&= c_0 + 2c_1 N(1 - 2^{-k}) = c_0 + 2c_1 N\Big(1 - \frac{999}{N}\Big) \\
&= c_0 + 2c_1 N - 1,998c_1 \in \Theta(N)
\end{aligned}$$

Another approach to solve this problem is by drawing the recursion tree and find the total work done by every node in the tree.

Will you want to pick up your worksheet later? Circle one: Yes / No
How confident do you feel with the material this week? Circle one: 1 / 2 / 3 / 4 / 5

2 Array Resizing

We implement `Queue` ADT using `ArrayQueue3` that `add` and `remove` methods work as expected. However, when we try to `add` a new element into the full `Queue`, we will make a new array with size of `size+1` and copy every element over.

(a) What are the valid runtime bound of `add` in each case? (choose **ALL** that apply)

Best case:

■ $\Omega(1)$ ■ $\Theta(1)$ ■ $O(1)$
☐ $\Omega(logN)$ ☐ $\Theta(logN)$ ■ $O(logN)$
☐ $\Omega(N)$ ☐ $\Theta(N)$ ■ $O(N)$

Worst Case:

■ $\Omega(logN)$ ☐ $\Theta(logN)$ ☐ $O(logN)$
■ $\Omega(N)$ ■ $\Theta(N)$ ■ $O(N)$
☐ $\Omega(N^2)$ ☐ $\Theta(N^2)$ ■ $O(N^2)$

Overall:

■ $\Omega(1)$ ☐ $\Theta(1)$ ☐ $O(1)$
☐ $\Omega(logN)$ ☐ $\Theta(logN)$ ☐ $O(logN)$
☐ $\Omega(N)$ ☐ $\Theta(N)$ ■ $O(N)$
☐ $\Omega(N^2)$ ☐ $\Theta(N^2)$ ■ $O(N^2)$

(b) If `ArrayQueue3` starts empty with an array of size 4, give a sequence of 6 operations that would perform the worst runtime (e.g., `add/remove/add/remove/add/remove` is a sequence of operations):

`add/add/add/add/add/add`

(c) What could be improved on `ArrayQueue3` to improve overall performances? (one sentence should be enough.)

Adding an element into the full queue will cost us $\Theta(N)$ run time. From 2(b), we can see that consecutive calls of `add` when queue is full will make the worst performance. Hence, we make a new array with size `size*2` instead of `size+1` when adding to the full queue.

(More in-depth explanation. Will not be tested)
Analyzing the overall performances of new `add` operation:

We will talk about the case where only `add`'s are called to demonstrate the worst case possible. Starting with an array of size 4, we know that the array will double its size when $size = 4, 8, 16, 32, 64, ..., N$ (assuming $N = 2^k$). For any other $size$, `add` will perform in constant time because resizing is unnecessary. So, the major contribution to the time complexity is $4 + 8 + 16 + 32 + 64 + ... + N = (2N - 1) - 2 - 1 = 2N - 4$. Therefore, the amortized cost of `add` is somewhat $\frac{2N-4}{N} = 2 - \frac{4}{N} \in \Theta(1)$. Note that, amortized analysis gives the average performance of the operation. That's why we divide by N.

3 Algorithm Analysis (Optional)

(a) $\Theta(\underline{N\quad\quad})$

```
static void addItUp(int N) {
    int x = 0;
    for (int i = 1; i <= N; i *= 2) {
        for (int j = 0; j < i; j++) {
            x++;
        }
        if (i % 2 == 1) {
            for (int j = 0; j < N; j++) {
                x++;
            }
        }
    }
    System.out.println(x);
}
```

Since values of `i` will be $1, 2, 4, 8, 16, ..., N$, we can see that `i % 2 == 1` is `true` only when `i=1`. Therefore, the for-loop inside if statement will only contribute $c_1 N$ amount of work.

For the second for-loop, `x++` will be executed for `i` times. Since `i` changes in each iteration, the worst-case runtime of this section can be modeled as $c_2(1+2+4+8+...+N) = c_2(2N-1)$. Formally, we can model this section as $\Sigma_{i=0}^{log_2 N}\Sigma_{j=1}^{2^i} c_2$ which simplifies to $\Sigma_{i=0}^{log_2 N} c_2 2^i = c_2(1+2+4+8+...+N)$.

Therefore, we can model the worst-case runtime of this whole algorithm as

$$c_2(2N-1) + c_1 N + c_0 \in \Theta(N)$$

where c_0, c_1, c_2 are some constants