1. Implement Promise.all Polyfill

```
function promiseAll(promises) {
    return new Promise((resolve, reject) => {
        let results = [];
        let completed = 0;

        promises.forEach((p, index) => {
            Promise.resolve(p).then(value => {
                results[index] = value;
                completed++;
                if (completed === promises.length) {
                    resolve(results);
                }
            }).catch(reject);
        });

        if (promises.length === 0) {
            resolve([]);
        }
    });
}
```

2. Limit Concurrent API Calls

```
async function limitedConcurrency(tasks, limit) {
    const results = [];
    let i = 0;
    const runNext = async () => {

        if (i >= tasks.length) return;

        const current = i++;

        results[current] = await tasks[current]();

        await runNext();

    };


    const runners = [];

    for (let j = 0; j < limit; j++) {

        runners.push(runNext());

    }


    await Promise.all(runners);

    return results;

}
```

3. Retry with Delay

```
function retry(fn, retries = 3, delay = 1000) {

    return new Promise((resolve, reject) => {

        const attempt = () => {

            fn()

                .then(resolve)

                .catch((err) => {

                    if (retries === 0) return reject(err);

                    retries--;

                    setTimeout(attempt, delay);

                });
        };

        attempt();

    });

}
```

4. Sleep Function (delay using Promise)

```
function sleep(ms) {

    return new Promise(resolve => setTimeout(resolve, ms));

}



async function run() {

    console.log("Start");

    await sleep(1000);

    console.log("End after 1 second");

}
```

5. Sequential Execution of Promises

```
async function runSequentially(tasks) {

    const results = [];

    for (let task of tasks) {

        results.push(await task());

    }

    return results;

}
```

6. Create a Timeout Wrapper

```
function withTimeout(promise, ms) {

    const timeout = new Promise((_, reject) =>
        setTimeout(() => reject(new Error("Timeout")), ms)
    );
    return Promise.race([promise, timeout]);
}
```

7. Chaining Promises

```
function chainPromises(funcs) {
    return funcs.reduce((p, fn) => p.then(fn), Promise.resolve());
}
```

8. Debounce Async Function (Promises)

```
function debounceAsync(fn, delay) {
    let timer;
    return (...args) => {
        clearTimeout(timer);
        return new Promise(resolve => {
            timer = setTimeout(() => resolve(fn(...args)), delay);
        });
    };
}
```

9. Promise Pool (Advanced)

Similar to question 2, but may also ask to cancel, pause, or add priorities.

Bonus: Real-World Coding Tasks

1. Load data from multiple APIs and display results in order of completion.

2. Fetch paginated data until no more pages are left.

3. Create a `cachePromise(fn)` that returns the cached result of an async function based on its arguments.