1. Diferencia entre interfaz y clase abstracta

R: Con la introducción de los default methods en Java la diferencia se volvió pequeña, principalmente podría decir que una clase abstracta sí puede tener constructor y una interfaz no, también el scope de los métodos cambia, en una interfaz no puedes tener métodos final, protected y private mientras que una clase abstracta no tiene limitaciones en ese aspecto, también en los campos que pueda tener, en una interfaz los campos no pueden ser no estáticos y no finales y por último diría que su uso por lo menos en Java una clase sólo puede heredar una clase pero tener múltiples implementaciones de interfaces así una clase puede heredar de diferentes fuentes si implementa varias interfaces y por clase abstracta sólo podría de una.

2. Diferencia entre LinkedList y ArrayList

R: La diferencia principal yo diría está en las implementaciones de cada una un ArrayList tiene la implementación de un array dinámico es decir la colocación de sus elementos son continuos en memoria y conforme crece va aumentando su espacio de forma dinámica, mientras que una LinkedList está implementada como una lista doblemente ligada es decir cada elemento es un nodo que apunta a otros nodos no necesariamente continuos en memoria, esto hace que cuando se trata de manipular la colección es mucho más eficiente que un ArrayList porque no tiene que hacer un shift de toda la colección ya que la memoria no es continua. Un ArrayList es más útil cuando se trata de guardar elementos y acceder a ellos, mientras que una LinkedList es más útil cuando se trata de manipular elementos.

3. Cómo funciona el garbage collector

R: El garbage collector es un proceso que administra la memoria de la JVM (para el caso de Java) en el cuál conforme el tiempo de vida de un objeto y su uso en los procesos de la JVM se va asignando a diferentes tipos de clasificaciones, dependiendo de las clasificaciones es cómo el garbage collector va eliminando los objetos que ya no se usan y va liberando memoria de la JVM.

4. Escribe la función POW sin usar multiplicación/división ni las funciones de las mismas.

R:
```
public static int pow(int base, int exponent){
    if (exponent <= 0){
      return 1;
    }
    int absoluteBase = 0;
    if(base < 0){
      absoluteBase = base * -1;
    } else {
      absoluteBase = base;
    }
    int result = absoluteBase;

    for( int i = 1; i<exponent; i++){
      int add= 0;
```

```
      for (int j = 0; j< absoluteBase; j++){
        add += result;
      }
      result =  add;
    }
    return (base < 0 && exponent % 2 == 1) ? -result : result;
  }
```

5. **PROBLEMA** ¿Cómo podemos solucionar este problema para cumplir con los 20ms? Explica cómo lo harías, qué tecnologías utilizarías y si es posible muestra la parte crucial de este código.

Lo que yo haría es hacer asíncronas las llamadas a las APIs que demoran 50ms en responder en vez de hacerlas de manera continua para poder alcanzar los 50TPS que pide el proveedor, esto lo haría utilizando la API de **Future** en Java y complementándola con **CompletableFuture** para hacer las llamadas asíncronas, junto con un tunning adecuado del ThreadPool que podría configurarse fácilmente en Spring con un bean y haciendo el método @Async y habilitando la asyncronia de métodos en Spring con @EnableAsync para delegar el manejo de los Executor a Spring. Si eso no fuera suficiente, trataría de validar las necesidades de éstas APIs que tardan 50ms y ver si se puede implementar un PATCH para acelerar los tiempos de respuestas pero esto dependería de las operaciones. Sin embargo, creo que habilitar los métodos asíncronos nos permitiría cumplir los 50TPS que pide el proveedor más allá de cumplir los 20ms por segundo y quizá llegaríamos a los 20ms. También es importante implementar un correcto uso de patrones de estabilidad como manejo de timeout, circuit brakers, manejo de carga, porque podríamos sobrecargar los tiempos de respuesta del lado de las APIs que tardan 50ms.

El código muy resumido y breve se vería así:

```
@Async
public CompletableFuture<CustomResponse> callApi(CustomRequest myRequest) throws
InterruptedException {
   String url = String.format("myurl...");
   CustomResponse results = restTemplate.getForObject(url, CustomResponse.class);
   //Logic ...
   return CompletableFuture.completedFuture(results);
}
```

La configuración de Spring sería muy similar a esta:

```
@SpringBootApplication
@EnableAsync
public class MyApplication {

   public static void main(String[] args) {
      // close the application context to shut down the custom ExecutorService
      SpringApplication.run(MyApplication.class, args).close();
   }

   @Bean
   public Executor taskExecutor() {
      ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
      executor.setCorePoolSize(10); //this properties should be tunned
      executor.setMaxPoolSize(20);
      executor.setQueueCapacity(500);
      executor.setThreadNamePrefix("apicallthread-");
      executor.initialize();
      return executor;
   }
```

Y el kickoff de las llamadas sería algo así

```
// asynchronus lookups
CompletableFuture<MyResponse> response1 = myService.callApi("apiOne");
CompletableFuture<MyResponse> response2 = myService.callApi("apiTwo");
CompletableFuture<MyResponse> response3 = myService.callApi("apoThree");

// wait until they are all done
CompletableFuture.allOf(response1,response2,response3).join();
logger.info("--> " + response1.get());
logger.info("--> " + response2.get());
logger.info("--> " + response3.get());
```