Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Sistemas e Industrial
Curso: Ingeniería de Software 1 (2016701)

Entrega 06 **Módulo de Testing** - Grupo 7 UNEmployed

Integrantes:
- ANGEL DAVID GÓMEZ PASTRANA
- DIEGO FELIPE CABRERA SALAMANCA
- PABLO LUNA GUZMÁN
- JAURI ESTEBAN CORTÉS CÁRDENAS

1. Para la implementación de las pruebas unitarias se utilizaron las librerías **Mocha** y **Chai**, ambas bastante utilizadas dentro del ecosistema de Node.js y compatibles con TypeScript mediante `ts-node`. **Mocha** actúa como *test runner*, permitiendo estructurar las pruebas en bloques `describe` e `it`, mientras que **Chai** proporciona un conjunto de funciones de aserción, empleando el estilo `expect`, que facilita la validación explícita de los resultados esperados. La ejecución de todas las pruebas del proyecto se realiza mediante el comando configurado en el script de NPM en la sección de Backend, definido como:

```
TypeScript
"test": "node ./node_modules/mocha/bin/mocha.js -r
ts-node/register tests/**/*.test.ts"
```

Esto dado que si no usamos la instalación local de mocha, Node intenta buscar una instalación global de esta y por tanto generaría problemas de portabilidad.

¿Y por qué no usamos Jest?

Jest es la librería más conocida para hacer test en Java, JavaScript y TypeScript pero Jest al instalarlo, nos traía varias vulnerabilidades de las cuales nos advirtió NPM y aunque no eran críticas, decidimos optar por Mocha con Chai, librerías las cuales no nos traían ninguna vulnerabilidad y además traen una sintaxis sencilla y explícita para los tests que debíamos hacer.

A continuación mostramos los test que se hicieron para algunas de las funciones que se hicieron por integrante:

Test angomezp:

- **Función Distancia de Haversine:**
  La función se usa para calcular la distancia entre dos coordenadas sobre una esfera de cierto radio (Asegura mayor precisión para los cálculos de distancia entre puntos del recorrido del entrenamiento)

TypeScript

```
it('should calculate distance between two points correctly', () => {
	const lat1 = 40.7128;
	const lon1 = -74.0060;
	const lat2 = 34.0522;
	const lon2 = -118.2437;

	const distance = haversineDistance(lat1, lon1, lat2, lon2);

	expect(distance).to.be.closeTo(3935.75, 0.1);
});

it('should return 0 for the same coordinates', () => {
	const distance = haversineDistance(40.7128, -74.0060, 40.7128,
-74.0060);
	expect(distance).to.equal(0);
});

it('should calculate short distances correctly', () => {
	const distance = haversineDistance(4.6097, -74.0817, 4.6107,
-74.0827);
	expect(distance).to.be.closeTo(0.157, 0.01);
});

it('should handle very short distances (less than 20 meters)', () => {
	const distance = haversineDistance(4.6097, -74.0817, 4.60972,
-74.08172);
	expect(distance).to.be.closeTo(0.00314, 0.0001);
});

it('should handle large distances', () => {
	const distance = haversineDistance(40.7128, -74.0060, -33.4489,
-70.6693);
	expect(distance).to.be.closeTo(8253.50, 0.1);
});

it('should return NaN for latitude out of valid range (> 90)', () => {
	const distance = haversineDistance(100, 0, 40, 0);
	expect(distance).to.be.NaN;
});

it('should return NaN for latitude out of valid range (< -90)', () => {
```

```
        const distance = haversineDistance(-100, 0, 40, 0);
        expect(distance).to.be.NaN;
});

it('should return NaN for longitude out of valid range (> 180)', () => {
        const distance = haversineDistance(40, 200, 40, 0);
        expect(distance).to.be.NaN;
});

it('should return NaN for longitude out of valid range (< -180)', () => {
        const distance = haversineDistance(40, -200, 40, 0);
        expect(distance).to.be.NaN;
});

it('should accept coordinates at valid boundaries', () => {
        const distance1 = haversineDistance(90, 0, -90, 0);
        const distance2 = haversineDistance(0, 180, 0, -180);
        expect(distance1).to.not.be.NaN;
        expect(distance2).to.not.be.NaN;
});
```

- Los tests de la función ***‘’haversineDistance”*** son importantes porque nos permiten comprobar que realmente está calculando bien las distancias y que no falla en situaciones clave. Se revisa que funcione con distancias grandes entre ciudades conocidas, que devuelve 0 cuando los puntos son exactamente los mismos y que siga siendo preciso cuando las distancias son muy cortas, donde normalmente aparecen errores numéricos. También se valida que la función sepa detectar coordenadas que no existen (como latitudes mayores a 90°) y devuelve `NaN` en esos casos, para evitar resultados falsos. Por último, se comprueba que los valores límite válidos sí los acepte sin problema. Todo esto hace que las pruebas no sean triviales, porque revisan tanto la precisión de los cálculos como el manejo de errores y de casos reales que podrían aparecer en el uso de la aplicación.
- **Función Calcular Calorias:**

TypeScript

```
it('should calculate calories for low speed (< 8 km/h)', () => {
        const calories = calculateCalories(5, 70, 1);
        expect(calories).to.equal(560);
});

it('should calculate calories for medium speed (8-12 km/h)', () => {
        const calories = calculateCalories(10, 70, 1);
        expect(calories).to.equal(700);
});
```

```
it('should calculate calories for high speed (> 12 km/h)', () => {
      const calories = calculateCalories(15, 70, 1);
      expect(calories).to.equal(840);
});

it('should calculate calories with different weight', () => {
      const calories1 = calculateCalories(10, 60, 1);
      const calories2 = calculateCalories(10, 80, 1);

      expect(calories1).to.equal(600);
      expect(calories2).to.equal(800);
});

it('should calculate calories with different time', () => {
      const calories1 = calculateCalories(10, 70, 0.5);
      const calories2 = calculateCalories(10, 70, 2);

      expect(calories1).to.equal(350);
      expect(calories2).to.equal(1400);
});

it('should return 0 for zero time', () => {
      const calories = calculateCalories(10, 70, 0);
      expect(calories).to.equal(0);
});

it('should handle exact boundary of 8 km/h', () => {
      const calories = calculateCalories(8, 70, 1);
      expect(calories).to.equal(700);
});

it('should handle exact boundary of 12 km/h', () => {
      const calories = calculateCalories(12, 70, 1);
      expect(calories).to.equal(840);
});

it('should return NaN for negative speed', () => {
      const calories = calculateCalories(-5, 70, 1);
      expect(calories).to.be.NaN;
});

it('should return NaN for zero weight', () => {
      const calories = calculateCalories(10, 0, 1);
      expect(calories).to.be.NaN;
});

it('should return NaN for negative weight', () => {
      const calories = calculateCalories(10, -70, 1);
```

```
        expect(calories).to.be.NaN;
});

it('should return NaN for negative time', () => {
        const calories = calculateCalories(10, 70, -1);
        expect(calories).to.be.NaN;
});
```

- Los tests de la función ***"calculateCalories"*** son útiles porque nos permiten verificar que el cálculo realmente cambia según la velocidad, el peso y el tiempo, que son las variables principales del modelo. Primero se revisan los tres rangos de velocidad (baja, media y alta) para asegurarnos de que la fórmula esté aplicando el factor correcto en cada caso. Luego se prueba cómo cambia el resultado cuando varía el peso y el tiempo, lo cual es clave porque estos valores afectan directamente las calorías quemadas. También se revisan casos límite como cuando el tiempo es cero, para que la función no devuelva un valor absurdo. Además, se prueban las velocidades exactas en los límites de 8 km/h y 12 km/h para confirmar que la lógica de las condiciones funciona bien justo en esos puntos. Finalmente, se validan entradas inválidas como velocidades negativas, pesos no válidos o tiempo negativo, esperando que la función devuelva NaN en lugar de cálculos incorrectos. Todo esto hace que las pruebas no sean triviales, porque evalúan diferentes escenarios reales y comprueban que la función sea robusta ante datos erróneos.

- **Funcion calcular Pace:**

TypeScript

```
it('should calculate pace correctly', () => {
        const pace = calculatePace(10);
        expect(pace).to.equal('6:00');
});

it('should handle very slow speeds', () => {
        const pace = calculatePace(2);
        expect(pace).to.equal('30:00');
});

it('should return "0:00" for zero speed', () => {
        const pace = calculatePace(0);
        expect(pace).to.equal('0:00');
});

it('should format seconds with leading zero', () => {
        const pace = calculatePace(12);
        expect(pace).to.equal('5:00');
```

```
});

it('should handle very high speeds', () => {
      const pace = calculatePace(20);
      expect(pace).to.equal('3:00');
});

it('should calculate pace with decimals correctly', () => {
      const pace = calculatePace(8.5);
      expect(pace).to.equal('7:03');
});

it('should handle speed very close to zero', () => {
      const pace = calculatePace(0.1);
      expect(pace).to.equal('600:00');
});

it('should handle extremely high speed', () => {
      const pace = calculatePace(100);
      expect(pace).to.equal('0:36');
});

it('should return "0:00" for negative speed', () => {
      const pace = calculatePace(-10);
      expect(pace).to.equal('0:00');
});

it('should handle speed that produces seconds close to 60', () => {
      const pace = calculatePace(11.11);
      expect(pace).to.equal('5:24');
});
```

- Estos tests buscan asegurar que la función ***"calculatePace"*** no solo haga la fórmula básica, sino que responda bien a casos que suelen romper este tipo de cálculos. Probamos una velocidad normal para verificar que lo básico funcione, y luego velocidades muy bajas o muy altas porque ahí es donde suelen salir errores con divisiones y formatos raros. También revisamos que los segundos queden con un cero adelante cuando toca, que los decimales no dañen el cálculo real del ritmo, y que valores extremos como casi cero, extremadamente altos o negativos no generen errores raros sino un resultado coherente. Incluso probamos una velocidad que produce segundos casi llegando a 60, para verificar que no se desborde el formato. En conjunto, estos tests ayudan a que la función sea confiable en situaciones reales y en casos límite.

Test dicabreras:

- Función lógica de entrenamiento: “calculateAvgSpeed”. Esta función hace parte de las estadísticas que se muestran en la grabación de un entrenamiento. Su implementación se encuentre en en el directorio /Backend/src/services/trainingServices.ts.

Esta función se encarga de retornar la velocidad promedio del entrenamiento grabado por un usuario.
Se pusieron a prueba los siguientes casos:

1. Calcular correctamente la velocidad promedio para un entrenamiento que registra 10 km en 1 hora.
2. Calcular correctamente la velocidad promedio para un entrenamiento que registra 5 km en 30 minutos.
3. Calcular correctamente la velocidad promedio para un entrenamiento que registra 0 segundos de actividad.

TypeScript

```
describe('calculateAvgSpeed', () => {

       it('should calculate average speed correctly for 10 km in 1 hour', ()
=> {

       const avg = calculateAvgSpeed(10, 3600); // 10 km, 3600s

      expect(avg).to.be.closeTo(10, 0.0001);

       });



       it('should calculate average speed correctly for 5 km in 30 minutes',
() => {

       const avg = calculateAvgSpeed(5, 1800); // 5 km, 1800s => 10 km/h

      expect(avg).to.be.closeTo(10, 0.0001);

       });



       it('should return 0 when totalSeconds is 0', () => {

       const avg = calculateAvgSpeed(5, 0);

       expect(avg).to.equal(0);

       });
```

```
});
```

- Función lógica de entrenamiento: “calculateMaxSpeed”. Esta función hace parte de las estadísticas que se muestran en la grabación de un entrenamiento. Su implementación se encuentre en en el directorio /Backend/src/services/trainingServices.ts.

Esta función se encarga de retornar la velocidad máxima en cualquier instante del entrenamiento grabado por un usuario.
Se pusieron a prueba los siguientes casos:

1. retorna correctamente la velocidad máxima de un array de velocidades tomada en distintos momentos del entrenamiento.
2. Retornar 0 cuando el array esta vacío, es decir, no hubo ningún registro de distancia y por tanto tampoco velocidad.
3. Determinar correctamente la velocidad para un array con un solo elemento. Retorna este único elemento.

TypeScript

```
describe('calculateMaxSpeed', () => {

    it('should return the maximum speed from array', () => {
      const speeds = [3.2, 7.5, 12.1, 9.9];
      const max = calculateMaxSpeed(speeds);
      expect(max).to.equal(12.1);
    });


    it('should return 0 for empty speeds array', () => {
      const max = calculateMaxSpeed([]);
      expect(max).to.equal(0);
    });


    it('should handle single-element array', () => {
```

```
      const max = calculateMaxSpeed([5.5]);
      expect(max).to.equal(5.5);
    });
  });
```

- Función lógica de gestión de entrenamientos: “saveTraining”. Esta función hace parte de las operaciones que se realizan con los entrenamientos una vez la grabación de este ha sido completada exitósamente. Su implementación se encuentra en en el directorio /Backend/src/controller/trainingController.ts.

Esta función permite guardar los entrenamientos verificando siempre que el usuario exista en el sistema y sus credenciales coincidan con las guardas en la base de datos.
Se puso a prueba el siguiente caso:

1. Debe guardar el entrenamiento de un usuario exitósamente
2. Gestionar apropiadamente el error cuando el correo del usuario no está registrado
3. Gestionar debidamente el error cuando el usuario no aparece en la base de datos
4. Rechaza el guardado de un entrenamiento cuando la distancia es 0 o no se registro ninguna ruta.
5. El guardado del entrenamiento falla

TypeScript

```
describe('TrainingController.saveTraining', () => {
  let controller: any;
  let originalGetRepository: any;
  let shouldThrowOnSave = false;

  before(() => {
      // Build fake repositories with minimal behavior required by the controller
    let coordId = 1;
    let routeId = 1;
    let trainingCounter = 1;
```

```
const fakeUser = {
  email: 'test@local',
  username: 'tester',
  names: 'Test',
  lastNames: 'User',
  age: 30
};

const fakeRepos = {
  User: {
    findOne: async (opts: any) => {
      const email = opts?.where?.email;
      if (email === fakeUser.email) return fakeUser;
      return null;
    }
  },
  Coordinate: {
    create: (obj: any) => ({ ...obj }),
    save: async (coord: any) => {
      coord.id = coordId++;
      return coord;
    }
  },
  Route: {
    create: (obj: any) => ({ ...obj }),
```

```
        save: async (route: any) => {
          route.id = routeId++;
          return route;
        }
      },
      Training: {
        create: (obj: any) => ({ ...obj }),
        save: async (t: any) => {
          if (shouldThrowOnSave) throw new Error('Simulated save error');
          t.counter = trainingCounter++;
          return t;
        }
      }
    };


        // Patch appDataSource.getRepository to return appropriate fake repository
    const ds = require('../../src/db_connection/config/dataSource');
    // Save original getRepository to restore later
    originalGetRepository = ds.appDataSource.getRepository;
    ds.appDataSource.getRepository = (entity: any) => {
      const name = entity && entity.name ? entity.name : String(entity);
      if (name.includes('User')) return fakeRepos.User;
      if (name.includes('Coordinate')) return fakeRepos.Coordinate;
      if (name.includes('Route')) return fakeRepos.Route;
      if (name.includes('Training')) return fakeRepos.Training;
```

```
      // Fallback repository with basic create/save methods
        return { create: (o: any) => ({ ...o }), save: async (o: any) => ({
...o }) };
    };

    // Now import the controller after the stub is in place
                                                  controller            =
require('../../src/db_connection/controller/TrainingController');
  });

  // Restore original getRepository to avoid interfering with other tests
  after(() => {
    try {
      const ds = require('../../src/db_connection/config/dataSource');
              if  (originalGetRepository)  ds.appDataSource.getRepository  =
originalGetRepository;
    } catch (e) {
      // ignore restore errors in test environment
    }
  });

  it('should save a training successfully', async () => {
    const req: any = {
      body: {
        userEmail: 'test@local',
        distance: 5000,
        duration: '00:25:00',
```

```
        avgSpeed: 12.0,
        maxSpeed: 15.0,
        rithm: 5.0,
        calories: 300,
        elevationGain: 10,
        trainingType: 'Running',
        route: [
          { latitude: 4.6, longitude: -74.0, altitude: 2550 },
          { latitude: 4.6001, longitude: -74.0002, altitude: 2552 }
        ],
        datetime: '2025-11-16T12:00:00Z'
      }
    };

    const res: any = {
      statusCode: 0,
      body: null,
      status(code: number) { this.statusCode = code; return this; },
      json(obj: any) { this.body = obj; return this; }
    };

    await controller.saveTraining(req, res);

    expect(res.statusCode).to.equal(201);
            expect(res.body).to.have.property('message', 'Training saved
successfully');
```

```
    expect(res.body).to.have.property('training');

    const t = res.body.training;

    expect(t).to.have.property('distance');

    expect(t.distance).to.equal(5000);

    expect(t).to.have.property('avgSpeed');

    expect(t.avgSpeed).to.equal(12.0);

  });


  it('should return 400 if userEmail is missing', async () => {

    const req: any = { body: { distance: 100 } };

     const res: any = { statusCode: 0, body: null, status(code: number) { this.statusCode = code; return this; }, json(obj: any) { this.body = obj; return this; } };

    await controller.saveTraining(req, res);

    expect(res.statusCode).to.equal(400);

  });


  it('should return 404 when user not found', async () => {

     const req: any = { body: { userEmail: 'notfound@local', route: [ { latitude: 4.6, longitude: -74.0, altitude: 2550 }, { latitude: 4.6001, longitude: -74.0002, altitude: 2552 } ] } };

     const res: any = { statusCode: 0, body: null, status(code: number) { this.statusCode = code; return this; }, json(obj: any) { this.body = obj; return this; } };

    await controller.saveTraining(req, res);

    expect(res.statusCode).to.equal(404);

  });
```

```
  it('should reject training when no route and zero distance', async () => {

     const req: any = { body: { userEmail: 'test@local', distance: undefined
} };

      const res: any = { statusCode: 0, body: null, status(code: number) {
this.statusCode = code; return this; }, json(obj: any) { this.body = obj;
return this; } };

    await controller.saveTraining(req, res);

       // Now controller rejects trainings with no valid route and zero
distance

    expect(res.statusCode).to.equal(400);

  });



  it('should return 500 when training repository save fails', async () => {

    // enable throwing in fake repo

    shouldThrowOnSave = true;

     const req: any = { body: { userEmail: 'test@local', route: [ { latitude:
4.6, longitude: -74.0, altitude: 2550 }, { latitude: 4.6001, longitude:
-74.0002, altitude: 2552 } ] } };

      const res: any = { statusCode: 0, body: null, status(code: number) {
this.statusCode = code; return this; }, json(obj: any) { this.body = obj;
return this; } };

    await controller.saveTraining(req, res);

    expect(res.statusCode).to.equal(500);

    // restore flag

    shouldThrowOnSave = false;

  });

});
```

Test jcortesca:

Para el módulo desarrollado por mi parte, se implementaron pruebas unitarias enfocadas en tres componentes principales del backend: las funciones del controlador de usuario (`registerUser` y `loginUser`) y la función de utilidades `calculateElevation`. El objetivo de estos tests fue garantizar que los controladores gestionaran correctamente todos los casos esperados, tanto positivos como negativos, y que la función de cálculo respondiera adecuadamente ante datos reales y casos límite. A continuación se presentan los describes correspondientes.

---

1. Test para `registerUser`

La función `registerUser` es responsable de registrar un nuevo usuario en el sistema. En este describe se probaron los siguientes escenarios:

- Registro exitoso cuando todos los campos están completos.
- Respuesta adecuada cuando faltan campos requeridos.
- Manejo correcto cuando el usuario ya existe en base de datos.
- Comportamiento apropiado ante errores internos inesperados.

```
            it("should register user and return 201 with no missing
fields", async () => {

            //cuerpo del request falso
            req.body = { ...baseBody};

            findOneStub.resolves(null);
            saveStub.resolves({ id:1 })

            await registerUser(req, res);

            expect(statusStub.calledWith(201)).to.be.true;
        });

          it("Should register user and return 201 if 'description' is
missig", async() => {

            //body base
            req.body = { ...baseBody};
            //quitar description
            delete req.body['description'];

            await registerUser(req, res);
```

```
            expect(statusStub.calledWith(201)).to.be.true;
        })

        requiredFields.forEach((field) => {
            it(`should return 400 if '${field}' is missing`, async () =>
{
                // clonar body base
                req.body = { ...baseBody };

                // borrar SOLO ese campo
                delete req.body[field];

                await registerUser(req, res);

                expect(statusStub.calledWith(400)).to.be.true;

                const response = jsonStub.firstCall.args[0];
                expect(response.missing).to.include(field);
            });
        });

    });
```

Explicación:
Estas pruebas son necesarias para cubrir los posibles escenarios de registro. Se valida que la función identifique datos faltantes, detecte duplicados en la base de datos y procese correctamente un registro válido. Además, se comprueba que la función devuelva los códigos HTTP correctos y mensajes coherentes.

2. Test para `loginUser`

La función `loginUser` valida credenciales del usuario y autentica el acceso. Para este describe se cubrieron casos clave como:

- Inicio de sesión exitoso.
- Email no encontrado.
- Contraseña incorrecta.
- Campos faltantes.
- Manejo de errores inesperados.

Fragmento del test:

```
it("should return 400 if 'email' is missing", async () => {

            const req: any = {

                body: {

                    password: "123"

                }

            };

            const jsonStub = sinon.stub();

               const statusStub = sinon.stub().returns({ json: jsonStub
});

            const res: any = {

                status: statusStub

            };

            await loginUser(req, res);



            expect(statusStub.calledWith(400)).to.be.true;

            const response = jsonStub.firstCall.args[0];

              expect(response.message).to.equal("Missing required field:
email");

        });



        it("should return 400 if 'password' is missing", async () => {

            const req: any = {

                body: {

                    email: "juan@perez.com"

                }
```

```
            };

            const jsonStub = sinon.stub();

               const statusStub = sinon.stub().returns({ json: jsonStub
});

            const res: any = {

                status: statusStub

            };

            await loginUser(req, res);



            expect(statusStub.calledWith(400)).to.be.true;

            const response = jsonStub.firstCall.args[0];

             expect(response.message).to.equal("Missing required field:
password");

        });



        it("should return 404 if user email not found", async () => {

            const req: any = {

                body: {

                    email: "pepe@perez.com",

                    password: "123"

                }

            };

            const jsonStub = sinon.stub();

               const statusStub = sinon.stub().returns({ json: jsonStub
});

            const res: any = {
```

```
            status: statusStub
        };

        fakeRepo.findOne.resolves(null);

        await loginUser(req, res);

        expect(statusStub.calledWith(404)).to.be.true;

        const response = jsonStub.firstCall.args[0];
        expect(response.message).to.equal("Invalid email");

    });

    it("should return 401 if password is incorrect", async () => {
        const req: any = {
            body: {
                email: "pepe@perez.com",
                password: "123"
            }
        };
        const jsonStub = sinon.stub();
           const statusStub = sinon.stub().returns({ json: jsonStub
});
        const res: any = {
```

```
            status: statusStub
        };

        sinon.stub(bcrypt, "compare").resolves(false);

        await loginUser(req, res);

        expect(statusStub.calledWith(404)).to.be.true;

        const response = jsonStub.firstCall.args[0];
        expect(response.message).to.equal("Invalid email");
    });

    it("should return 500 on internal server error", async () => {
        const req: any = {
            body: {
                email: "pedro@pecas.com",
                password: "123"
            }
        };
        const jsonStub = sinon.stub();
           const statusStub = sinon.stub().returns({ json: jsonStub
});
        const res: any = {
            status: statusStub
```

```
            };

            fakeRepo.findOne.throws(new Error("DB error"));



            await loginUser(req, res);

            expect(statusStub.calledWith(500)).to.be.true;



            const response = jsonStub.firstCall.args[0];

            expect(response.message).to.equal("Internal server error");

        });
```

Explicación:

Las pruebas de este bloque verifican que el proceso de autenticación funcione de manera robusta ante situaciones reales: credenciales correctas, errores del usuario (correo incorrecto, contraseña errónea) y errores del servidor. Es fundamental asegurar que la función sea segura, devuelva códigos HTTP adecuados y no filtre información sensible.

3. Test de la función `calculateElevation`

Esta función toma una serie de coordenadas y calcula la diferencia máxima de altitud en un recorrido. Se probaron:

- Cálculo correcto con altitudes variadas.
- Casos donde todas las altitudes son iguales.
- Coordenadas sin altitud.
- Altitudes negativas.
- Arreglos con un solo elemento.
- Garantizar que nunca retorne un valor negativo.

```
describe("calculateElevation", () => {



        it("should return elevation difference for normal altitudes",
() => {

            const coords = [
```

```
                { latitude: 0, longitude: 0, timestamp: 0, altitude:
100 },

                { latitude: 0, longitude: 0, timestamp: 0, altitude:
150 },

                { latitude: 0, longitude: 0, timestamp: 0, altitude:
120 }

            ];

            const result = calculateElevation(coords);

            expect(result).to.equal(50);

        });



        it("should return 0 when all altitudes are equal", () => {

            const coords = [

                { latitude: 0, longitude: 0, timestamp: 0, altitude: 80
},

                { latitude: 0, longitude: 0, timestamp: 0, altitude: 80
},

                { latitude: 0, longitude: 0, timestamp: 0, altitude: 80
}

            ];

            const result = calculateElevation(coords);

            expect(result).to.equal(0);

        });



        it("should ignore undefined altitudes", () => {

            const coords = [

                { latitude: 0, longitude: 0, timestamp: 0, altitude:
100 },
```

```
                { latitude: 0, longitude: 0, timestamp: 0 }, // sin
altitud

                { latitude: 0, longitude: 0, timestamp: 0, altitude: 90
}

            ];

            const result = calculateElevation(coords);

            expect(result).to.equal(10);

        });



        it("should work with negative altitudes", () => {

            const coords = [

                { latitude: 0, longitude: 0, timestamp: 0, altitude:
-50 },

                { latitude: 0, longitude: 0, timestamp: 0, altitude:
-10 },

                { latitude: 0, longitude: 0, timestamp: 0, altitude:
-30 }

            ];

            const result = calculateElevation(coords);

            expect(result).to.equal(40);

        });



        it("should return 0 when array has only one coordinate", () =>
{

            const coords = [

                { latitude: 0, longitude: 0, timestamp: 0, altitude:
123 }

            ];
```

```
            const result = calculateElevation(coords);

            expect(result).to.equal(0);

        });



        it("should never return negative elevation", () => {

            const coords = [

                { latitude: 0, longitude: 0, timestamp: 0, altitude: 50
},

                { latitude: 0, longitude: 0, timestamp: 0, altitude: 50
}

            ];

            const result = calculateElevation(coords);

            expect(result).to.equal(0);

        });
```

Explicación:

Esta función se emplea para estimar la ganancia de elevación en un entrenamiento. Los tests permiten asegurar que trabaja correctamente incluso con datos incompletos, altitudes negativas, límites y casos triviales. También confirman que el resultado siempre sea lógico y no negativo, reforzando la fiabilidad del cálculo.

Test palunag: