Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 13-02-2024

Last Name: ....
First Name: ....
Student ID: ....
Signature: ....

<u>**Exam duration: 2 hours**</u>

Students can use a pen or a pencil for answering questions.

Students are NOT permitted to use books, course notes, calculators, mobile phones, and similar connected devices.

Students are NOT permitted to copy anyone else's answers, pass notes amongst themselves, or engage in other forms of misconduct at any time during the exam.

Writing on the cheat sheet is NOT allowed.

Exercise 1: _____ Exercise 2: _____ Exercise 3: _____

Exercise 1 (12 points)

You are required to provide an implementation of **PoliBikes** an electric bike-sharing support system that features three types of objects: (1) bikes; (2) bike stations; (3) bike station networks.

First, all bikes feature an initial location, color, number of brakes, and number of gears and must be uniquely identified. Also, there exist three types of bikes to be supported by the system, namely: (a) regular electric bike (that features an electric charge level); (b) tandem bike (which features multiple seats); and (c) cargo bike (which features a capacity load for the cargo).

Second, the bike stations are designed to host bikes of any type and are featured by a maximum capacity and a geographical location. There are two possible sub-types of bike stations: (a) electric-only bike stations (characterized by a maximum power volume); (b) regular-only bike stations (characterized by remaining free slots to park a bike).

Third, finally, the bike network manages the construction and operation of the network of bike stations and provides additional functions to (a) analyze the load of the network and (b) compute bike return adjustments to keep the network balanced.

An *initial* class diagram of the reference implementation is reported in Fig. 1. NOTE: the class diagram merely features class member attributes and is *deliberately incomplete*. Specifically, the diagram does not contain the signatures of all methods for all classes, nor does it address the visibility and scope of the member attributes for the various classes, and finally, it does not feature **getters and setters**.
Under the aforementioned definitions, your task is to:

1. complete the class diagram in Fig. 1 to provide: (a) the visibility of member attributes for all classes, using a **+** for `public`, a - for `private`, and a **#** for `protected` attributes;

2. complete the class diagram in Fig. 1 to provide the correct signatures (and only the signatures) of all necessary member functions of classes `Bike`, `BikeStation`, and `BikeNetwork` to address:

   (a) adding objects (for `BikeStation` and `BikeNetwork` only);

   (b) removing objects (for `BikeStation` and `BikeNetwork` only);

   (c) print object characteristics (for all).

   HINT: You must account appropriately for the required inheritance mechanisms and link the three classes. For example, all bikes must have a unique ID and other characteristics. Still, the concept of bike needs to remain `abstract`, while all classes shall have `protected` members as appropriate for their planned sub-classes; also, each class shall have `virtual` function or even pure virtual function specifications as appropriate for their planned sub-classes and operations;

3. provide the constructor of the `BikeNetwork` data structure, which receives as input a vector of `std::shared_ptr` s to `BikeStation` objects, and makes the appropriate initialisation, **printing out an error message** if

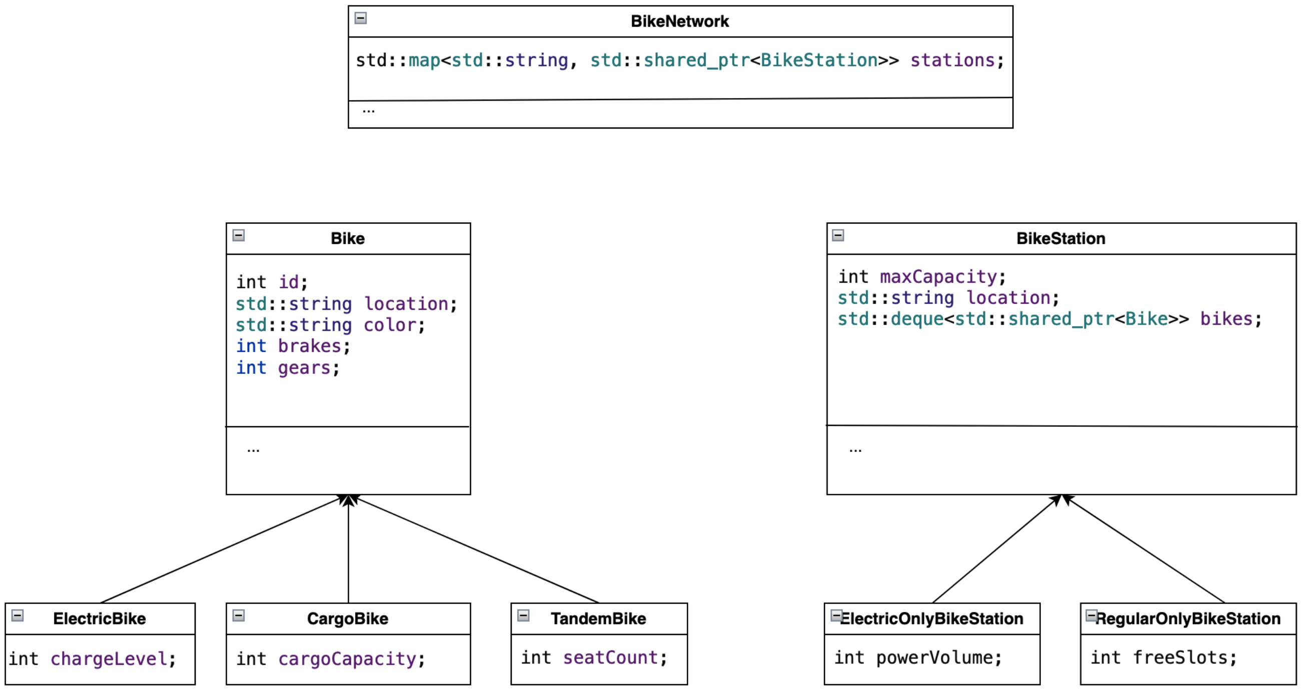

Figure 1: Sample class diagram of the basic data handled by the intended solution.

inconsistent pointer data is received. HINT: Each station is added to the `Stations` data structure within a `std::map`, using station location as the key.

Moreover, provide the implementation of the following methods:

4. `int BikeNetwork::getTotalBikesInNetwork() const` which operates a count of all the bikes currently in the system.

5. `std::vector<std::pair<std::string, int>> BikeNetwork::balanceNetwork() const` which, as previously specified, analyses the target `BikeNetwork` object and studies whether it is balanced, meaning that the function (a) calculates the average number of bikes per station; (b) checks whether all stations contain that exact number of bikes and if not, (c) finds the stations which are above or below average and 'adjusts' the count of bikes for those stations with any necessary removals/additions, storing such 'adjustment' operations as a vector of `std::pair<std::string, int>` where the first member of the pair is the address of the `station`, while the second member of the pair is the number of additions (positive number) or removals (negative number) to be operated for that station;

Finally, provide and discuss:

6. the **worst-case** complexity of all the methods you have implemented;

7. the cases of `dynamic-binding` emerging from the implementation. Specifically, identify what cases exist in the implementation and describe the use of any virtual methods emerging from those cases.

Solution 1

1. completing the class diagram with member attribute scope visibility;

   - Bike (Base Class) − > `int id`: Protected. Visible to the Bike class and its subclasses but not outside.
   - Bike (Base Class) − > all the attributes `std::string location`; `std::string color`; `int brakes`; `int gears`; shall be private, accessible only through getters and setters.
   - ElectricBike, TandemBike, CargoBike (Subclasses) − > are characterised by attributes `int chargeLevel`, `int cargoCapacity`, and `int seatCount`;

- BikeStation (Base Class) − > **int** maxCapacity: Protected. Accessible within BikeStation and its subclasses, but not from outside. std::string location: Protected. Same visibility as maxCapacity. std::deque<std::shared_ptr<Bike>> bikes: Protected. Accessible in BikeStation and its subclasses. This is where the bikes at each station are stored.
- ElectricOnlyBikeStation, RegularBikeOnlyStation (Subclasses) − > No new member attributes are introduced in these subclasses. They inherit and can access the protected members of BikeStation.
- std::map<std::string, std::shared_ptr<BikeStation>> stations − > Private. Accessible only within the BikeNetwork class. This map holds the bike stations in the network, keyed by their location.

2. Concerning the Bike class:

   - **virtual void** displayFeatures() **const** = 0; the method represents the constriction of the class to be abstract. Other methods to add or remove are demanded of the subclasses.

   Concerning the BikeStation class:

   - **virtual bool** addBike(std::shared_ptr<Bike> bike);
   - **bool** removeBikeById(**int** bikeId);
   - **virtual void** displayFeatures() **const** override;

   Concerning the BikeNetwork class:

   - **void** addStation(std::shared_ptr<BikeStation> station, **const** std::string& location);
   - **bool** removeBikeFromNetwork(**int** bikeId);
   - **void** displayNetworkDetails() **const**;

3. the constructor shall accept an initial list of bike stations and, assuming each station has a unique location that can be used as a key, will use it to perform a direct insertion. Specifically:

```
BikeNetwork(const std::vector<std::shared_ptr<BikeStation>>& initialStations) {
      for (const auto& station : initialStations) {
         if (!station) {
            std::cerr << "Error: Null station pointer encountered." << std::endl;
            continue; // Skip this station and continue with the next
         }
         // Assuming each station has a unique location that can be used as a key
         std::string location = station->getLocation();
         stations[location] = station;
      }
   }
```

4. the method operates a simple count of all elements in the system, using Station iterators opportunistically, specifically:

```
int BikeNetwork::getTotalBikesInNetwork() const {
␣␣␣␣int totalBikes = 0;
␣␣␣␣for (const auto& stationPair : stations) {
␣␣␣␣const auto& station = stationPair.second;
␣␣␣␣totalBikes += station->getBikeCount();
␣␣␣␣}
return totalBikes;
}
```

5. the method is again a mere scan of the elements in the system to compute the rearrangement of elements that restore balance across the system stations. A **for** loop is operated to perform the lookup and a std::vector is used to store the re-arrangements. Specifically:

```
std::vector<std::pair<std::string, int>> BikeNetwork::balanceNetwork() {
    // Calculate the total number of bikes across all stations in the network.
    int totalBikes = getTotalBikesInNetwork();
    // Get the number of stations in the network.
    int numStations = stations.size();
    // Calculate the average number of bikes per station.
    int averageBikes = totalBikes / numStations;
    // Prepare a vector to hold the adjustments needed for each station.
    std::vector<std::pair<std::string, int>> adjustments;
    // Iterate through each station in the network.
    for (const auto& stationPair : stations) {
        // Obtain a reference to the current station.
        const auto& station = stationPair.second;
        // Determine the current number of bikes at this station.
        int currentBikeCount = station->getBikeCount();

        // Calculate how many bikes need to be added or removed to reach the average.
        // If the station has more bikes than the average, this number will be negative.
        int adjustment = averageBikes - currentBikeCount;

        // Ensure that the adjustment does not exceed the station's capacity.
        // This prevents trying to add more bikes than the station can hold.
        adjustment = std::min(adjustment, station->getMaxCapacity() - currentBikeCount);

        // Add the station's identifier and the adjustment to the adjustments vector.
        adjustments.emplace_back(stationPair.first, adjustment);
    }

    // Return the vector containing the adjustments for each station.
    return adjustments;
}
```

6. Both `getTotalBikesInNetwork()` and `balanceNetwork()` have a linear complexity in terms of the number of stations in the network. These methods are efficient when the number of stations `S` is relatively small. However, as the network grows larger, the time taken by these methods will increase linearly with the number of stations. Specifically, the two methods we implemented are `BikeNetwork::getTotalBikesInNetwork()` and `BikeNetwork::balanceNetwork()` for which, the individual computational complexity can be elaborated as follows:

    - `BikeNetwork::getTotalBikesInNetwork() const;` — the method computes the total number of bikes across all stations in the network. First the method operates an iteration through each station, so If there are s stations, this iteration is `O(s)`. Subsequently, the method operates an access operation to the bike count of each station, so for each station, the method calls a `getBikeCount()` method or similar to perform a count of all bikes in the system, regardless of their specific sub-type. Assuming this method is `O(1)` (as it should typically be for a simple count retrieval), the overall complexity remains dominated by the number of stations. In summary, the worst-case Complexity of `getTotalBikesInNetwork()` is O(s).
    - `std::vector<std::pair<std::string, int>> BikeNetwork::balanceNetwork();` the method essentially aims to calculate the adjustments needed to balance the bikes across all stations. As per the exercise text, the method first calculates the total number of bikes and average, which involves a single pass through all stations (which we already established as `O(s)`) and subsequently carrying simple arithmetic operations, which are `O(1)`. Further on, the method iterates through each station again and for each the method computes the difference between the current bike count and the average, this is a constant-time operation. Finally, constructing the adjustments vector populates a vector with adjustment information for each station. This operation is `O(s)`, as it involves inserting s elements into the vector. In summary, the worst-case Complexity of `balanceNetwork()` is also O(s). The complexity is primarily due to iterating through all the stations twice, once for calculating the total and average and once for determining any adjustments needed.

7. dynamic-binding in this simple implementation reflects the hierarchical structure of the Bike class hierarchy as well as the BikeStation hierarchy. Specifically:

   - The Bike class features a pure virtual function `displayFeatures()` and is therefore an abstract class. Consequently, all subclasses like ElectricBike, TandemBike, and CargoBike override the `displayFeatures()`. When this function is called on a pointer or reference to Bike, the actual function invoked will be the one corresponding to the type of the object that the pointer or reference refers to (e.g., ElectricBike, TandemBike, or CargoBike). Concerning dynamic binding, the decision about which displayFeatures method to call is made at runtime.
   - The BikeStation class has a virtual method **`virtual bool`** `addBike(std::shared_ptr<Bike> bike)`, which is overridden in all its subclasses (e.g., ElectricOnlyBikeStation and RegularBikeOnlyStation, as per specification). At that point, when the `addBike` method is called on a pointer or reference to BikeStation, the actual method invoked depends on the object's actual subclass type. This is again dynamic-binding.

Exercise 2 (14 points)

You have to implement a **parallel program** that verifies if the matrix is *row diagonally dominant* and that computes its *lumped matrix*.
Given a matrix $A \in \mathbb{R}^{n \times n}$, we say that A is strictly row diagonally dominant if

$$|A(i,i)| > \sum_{i \neq j} |A(i,j)| \quad \forall i = 1, \ldots, n,$$

and we define the lumped matrix $M \in \mathbb{R}^{n \times n}$ as:

$$M(i,i) = \sum_{j=1}^{n} A(i,j) \quad i = 1, \ldots, n.$$

You have to implement a main file that calls the following two functions:

```
int checkDiagonalDominance(const la::dense_matrix & local_A, int n,int local_n, int rank);

la::dense_matrix computeLumpedMass(const la::dense_matrix &local_A, int n, int local_n, int rank);
```

The first function verifies if the matrix A is *row diagonally dominant.* The routine returns 1 if the matrix is row diagonally dominant, 0 otherwise. If one of the core finds a row that is not diagonally dominant, then the whole routine must return 0 via a proper reduction operation.
The second function computes the lumped mass matrix.
Furthermore, you can assume that:

1. The full matrix A is read from *rank* 0 by accessing a file; the file-name is passed through the command-line as a parameter.
2. Assume that the number of rows in the matrix is a multiple of the number of cores.
3. The lumped matrix M must be known by all the processors.

The initial part of the implementation is provided below:
main.cpp

```
#include <fstream>
#include <iostream>

#include <mpi.h>

#include "dense_matrix.hh"

int
main (int argc, char *argv[])
{
  MPI_Init (&argc, &argv);

  int rank (0), size (0);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  la::dense_matrix full_A;

  if (rank == 0)
  {
    std::ifstream first (argv[1]);
    full_A.read (first);
  }

  /*
```

```
   * YOUR CODE GOES HERE
   */


  MPI_Finalize ();
  return 0;
}
```

Header file for the helper class dense_matrix.h

```
#ifndef DENSE_MATRIX_HH
#define DENSE_MATRIX_HH

#include <istream>
#include <vector>

namespace la // Linear Algebra
{
  class dense_matrix final
  {
    typedef std::vector<double> container_type;

  public:
    typedef container_type::value_type value_type;
    typedef container_type::size_type size_type;
    typedef container_type::pointer pointer;
    typedef container_type::const_pointer const_pointer;
    typedef container_type::reference reference;
    typedef container_type::const_reference const_reference;

  private:
    size_type m_rows = 0, m_columns = 0;
    container_type m_data;

    size_type
    sub2ind (size_type i, size_type j) const;

  public:
    dense_matrix (void) = default;

    dense_matrix (size_type rows, size_type columns,
                  const_reference value = 0.0);

    explicit dense_matrix (std::istream &);

    void
    read (std::istream &);

    void
    swap (dense_matrix &);

    reference
    operator () (size_type i, size_type j);
    const_reference
    operator () (size_type i, size_type j) const;

    size_type
    rows (void) const;
    size_type
    columns (void) const;
    dense_matrix
    transposed (void) const;

    pointer
    data (void);
    const_pointer
    data (void) const;

    void
    print (std::ostream& os) const;

    void
    to_csv (std::ostream& os) const;
  };

  dense_matrix
  operator * (dense_matrix const &, dense_matrix const &);

  void
  swap (dense_matrix &, dense_matrix &);
}

#endif // DENSE_MATRIX_HH
```

Solution 2

A sample of the solution also containing some indicative content for the argumentation of the main is reported below.

```
#include <fstream>
#include <iostream>

#include <mpi.h>

#include "dense_matrix.hh"

int checkDiagonalDominance(const la::dense_matrix & local_A, int n, int local_n, int rank);
la::dense_matrix computeLumpedMass(const la::dense_matrix &local_A, int n, int local_n, int rank);

int main (int argc, char *argv[])
{
  MPI_Init (&argc, &argv);

  int rank (0), size (0);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  la::dense_matrix full_A;

  if (rank == 0)
  {
    std::ifstream first (argv[1]);
    full_A.read (first);
  }

  unsigned n;
  if (rank == 0){
      n = full_A.rows();
  }

  // get dimensions
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // partition the matrix
  unsigned local_n = n / size;

  // declare local chunks
  la::dense_matrix local_A(local_n,n);

  // spread all the chunks of the matrix
  MPI_Scatter(full_A.data(), local_n*n, MPI_DOUBLE,
           local_A.data(), local_n*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // check diagonal dominance
  int out = checkDiagonalDominance(local_A, n, local_n, rank);

  // declare lumped mass matrix
  la::dense_matrix global_M(n,n);

  // get lumped mass matrix
  global_M = computeLumpedMass(local_A, n, local_n, rank);


  MPI_Finalize();
  return 0;
}

int checkDiagonalDominance(const la::dense_matrix & local_A, int n, int local_n, int rank)
{
  // default return value
  int flag = 1;

  for (unsigned i=0; i < local_n; i++){

      // buffer for the diagonal term
      double diagBuffer = 0.0;
      int col = i+rank*local_n;

      // sum off diagonal terms
      for (unsigned j=0; j < col; j++){
            diagBuffer += local_A(i,j);
      }
      for (unsigned j=col+1; j < n; j++){
            diagBuffer += local_A(i,j);
      }

      // check diagonal dominance
      if (diagBuffer >= local_A(i,col) ) {
         flag = 0;
         break;
      }
  }

  MPI_Allreduce(MPI_IN_PLACE, &flag, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);

  return flag;
```

```

}

la::dense_matrix computeLumpedMass(const la::dense_matrix &local_A,
        int n, int local_n, int rank)
{
  la::dense_matrix global_M(n,n);

  // compute the sum
  la::dense_matrix local_M(local_n,n,0.0);

  for (unsigned i=0; i < local_n; i++){
        for (unsigned j=0; j < n; j++){
            local_M(i, i+rank*local_n) += local_A(i,j);
        }
  }

  MPI_Allgather(local_M.data(), local_n*n, MPI_DOUBLE,
               global_M.data(), local_n*n, MPI_DOUBLE, MPI_COMM_WORLD);

  return global_M;
}
```

Exercise 3 (4 points)

The provided code implements the simulation of animal growth, by instancing classes `Animal`, `Mammal`, and `Bird`.

The `main` function serves as a practical demonstration of the system functionalities. Animals can be declared and they can grow. While growing, their characteristics change.

After carefully reading the code, you have to answer the following questions. Please make sure to mark the answers clearly on the sheets. Moreover, it is mandatory for you to **develop your solution motivating the results** you achieved on the sheets. **Only providing the final answers is not enough** to obtain points in this section.

1. What is the value of attribute `eagle.furDensity` on line 13?
2. What is the value of variable `tiger.age` when it is displayed on line 25?
3. What is the value of variable `eagle.wingSpan` when it is displayed on line 26?
4. What is the value of variable `lion.age` when it is displayed on line 31?

Provided source code:

- **Animal.h**

```
#ifndef ANIMAL_H
#define ANIMAL_H

class Animal {
protected:
    int age;

public:
    Animal(int initialAge);

    virtual void grow();
    virtual int getMaxAge() const;
    virtual void display() const;
};

#endif
```

- **Animal.cpp**

```
#include "Animal.h"
#include <iostream>

Animal::Animal(int initialAge) : age(initialAge) {}

void Animal::grow() {
    if (age != -1) {
        age++;

        if (age > getMaxAge()) {
            age = -1;
        }
    }
}

int Animal::getMaxAge() const {
    return -1;
}

void Animal::display() const {
    std::cout << "Animal - Age: " << age << "\n";
}
```

- **Mammal.h**

```cpp
#ifndef MAMMAL_H
#define MAMMAL_H

#include "Animal.h"

class Mammal : public Animal {
private:
    int furDensity;

public:
    Mammal(int initialAge, int initialFurDensity);

    void grow() override;
    int getMaxAge() const override;
    void display() const override;
};

#endif
```

- **Mammal.cpp**

```cpp
#include "Mammal.h"
#include <iostream>

Mammal::Mammal(int initialAge, int initialFurDensity) : Animal(initialAge), furDensity(
     initialFurDensity) {}

void Mammal::grow() {
    if (age != -1) {
        Animal::grow();

        if (age <= 10) {
            furDensity += 3;
        } else {
            furDensity -= 5;

            if (furDensity < 0) {
                furDensity = -1;
                age = -1;
            }
        }
    } else {
        furDensity = -1;
    }
}

int Mammal::getMaxAge() const {
    return 20;
}

void Mammal::display() const {
    std::cout << "Mammal - Age: " << age << ", Fur Density: " << furDensity << "\n";
}
```

- **Bird.h**

```cpp
#ifndef BIRD_H
#define BIRD_H
```

```
#include "Animal.h"

class Bird : public Animal {
private:
    int wingSpan;

public:
    Bird(int initialAge, int initialWingSpan);

    void grow() override;
    int getMaxAge() const override;
    void display() const override;
};

#endif
```

- **Bird.cpp**

```
#include "Bird.h"
#include <iostream>

Bird::Bird(int initialAge, int initialWingSpan) : Animal(initialAge), wingSpan(initialWingSpan) {}

void Bird::grow() {
    if (age != -1) {
        Animal::grow();

        if (age <= 5) {
            wingSpan += 2;
        } else {
            wingSpan -= 1;

            if (wingSpan < 0) {
                wingSpan = -1;
                age = -1;
            }
        }
    } else {
        wingSpan = -1;
    }
}

int Bird::getMaxAge() const {
    return 10;
}

void Bird::display() const {
    std::cout << "Bird - Age: " << age << ", Wing Span: " << wingSpan << "\n";
}
```

- **main.cpp**

```
#include <iostream>
#include "Mammal.h"
#include "Bird.h"

int main() {
    Mammal lion(3, 10);
    Mammal tiger(8, 15);
    Bird eagle(2, 5);

    std::cout << "Initial state:\n";
    lion.display();
    tiger.display();
    eagle.display();

    std::cout << "\nGrowing...\n";
    for (int i = 0; i < 10; ++i) {
        int step_number = i + 1;
        lion.grow();
        tiger.grow();
        eagle.grow();

        if ((step_number) % 7 == 0) {
            std::cout << "\nDisplaying after " << step_number << " steps:\n";
            lion.display();
            tiger.display();
            eagle.display();
        }
    }

    std::cout << "\nDisplaying at the end:\n";
    lion.display();
    tiger.display();
    eagle.display();

    return 0;
}
```

Solution 3

1. Being of class Bird, it has no attribute `furDensity`.
2. **-1**.
   The tiger has lost all its fur and is therefore dead.
3. **7**.
   Starts at 5. For 3 times (from age 2 to age 5) it is incremented by 2, for a total of +6, then it is decremented by 1 each time for 4 times, for a total of -4.
4. **13**.
   Grew regularly: was 3 at the beginning, grew for 10 iterations.