Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna, Prof. Damian Tamburri

Date: 19-07-2024

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

Exam duration: 2 hours and 30 minutes

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)

A Service Value Network (SVN) represents the interconnections between a number of independent actors (i.e., the participants) cooperating in co-creating value by exchanging service requests and services rendered. Such actors are participants in an SVN as either humans or organizations exchanging positive (e.g., a 'buy' transaction), negative (e.g., a 'sell' transaction), or neutral (e.g., a fair goods exchange) value.

In operational terms, an SVN is defined as a graph tailored for modeling actor nodes and service relationships and calculating their value impacts. You are required to provide an implementation of **PoliSeVeN**, a system to support the specification and operation of SVNs. Specifically, you shall work on the implementation of a `ServiceValueNetwork` class featuring a list of `ServiceNode` elements defined by: (a) their own `data` attribute—a string representing the node name immediately followed by its unique ID (a positive integer starting from 1); (b) their own `value`—specifically, a number that can be positive, negative or zero; (c) a list of neighbors `nodes` with which each node has a value transaction in progress. NOTE: all nodes are at the same level of abstraction, and a relation between nodes shall reflect a transaction between such nodes.

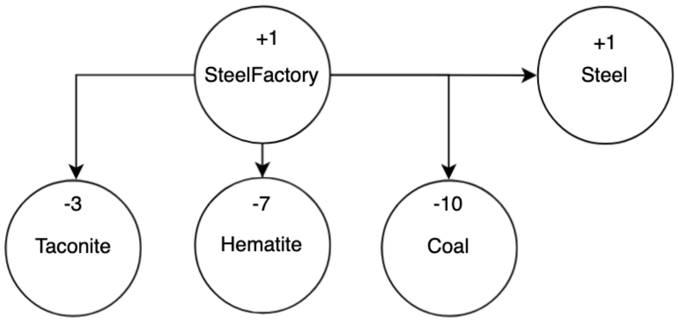


Figure 1: SVNs for PoliSeVeN, a simple example featuring one factory node and three ingredient nodes with a clear negative relationship in-between, and yielding a positive (production) result of 1 unit of steel.

As implicit in the above, we assume that the nodes always carry the same transactional value. For example, imagine a factory X that needs to manufacture iron from iron ore of some kind (rocks and minerals from which metallic iron can be economically extracted, such as hematite, taconite, or scrap iron) and carbon from coking coal following a specific recipe reflected by a precise set of (negative) SVN relations. In this case, all the ingredients would be negative-valued `ServiceNode`s, which reflects a negative ('sell') value in their relation with the factory X `ServiceNode`. A simple version of the aforementioned scenario is depicted in Fig. 1. In this scenario, the `SteelFactory` node, following a specific steel recipe, transacts 10-value worth of coal, 7-value worth of Hematite,

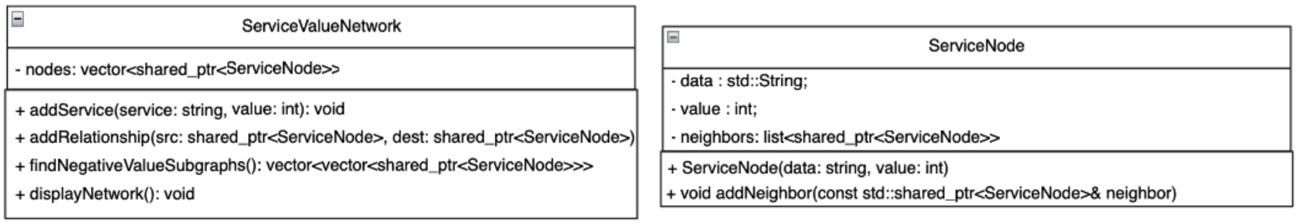

ServiceValueNetwork
- nodes: vector<shared_ptr<ServiceNode>>
+ addService(service: string, value: int): void
+ addRelationship(src: shared_ptr<ServiceNode>, dest: shared_ptr<ServiceNode>)
+ findNegativeValueSubgraphs(): vector<vector<shared_ptr<ServiceNode>>>
+ displayNetwork(): void
ServiceNode
- data : std::String;
- value : int;
- neighbors: list<shared_ptr<ServiceNode>>
+ ServiceNode(data: string, value: int)
+ void addNeighbor(const std::shared_ptr<ServiceNode>& neighbor)

Figure 2: a simplistic class diagram for PoliSeVeN.

and 3-value worth of Taconite, and produces 1-value worth of Steel. An *initial* class diagram of the reference implementation for PoliSeVeN is reported in Fig. 2. NOTE: `get`ters and `set`ters are omitted from the diagram for the sake of brevity but are available for you to use if needed.

Under the aforementioned definitions, your task is to implement the following methods of `ServiceValueNetwork`:

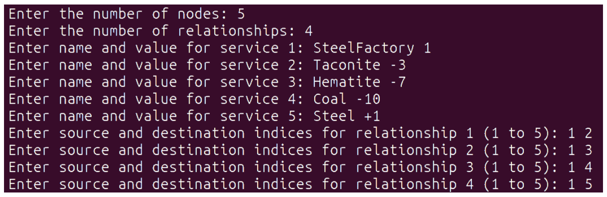
1. `void addService(const string& service, int value)` which adds a node into the network.
2. `void addRelationship(const shared_ptr<ServiceNode>& src, const shared_ptr<ServiceNode>& dest)` which adds a transactional service value relationship between two nodes. NOTE: you can assume the `addNeighbor(const shared_ptr<ServiceNode>& neighbor)` method is available for you to use in the scope of adding a relationship between a node with destination the `neighbor` node passed as parameter.

After implementing the above methods, you are required to:

3. provide the constructor of the `ServiceValueNetwork` class, which initializes the network based on specified numbers of nodes and relationships, **the aforementioned numbers are given as parameters to the constructor but the correct data to initialize nodes and relationships needs to be prompted to the user for input**. Fig. 3 shows an example list of appropriate user prompts and inputs, corresponding to the graph described in Fig. 1. All anomalous situations in this process shall **print out an error message**, e.g., if inconsistent data is provided. NOTE: to reduce re-allocations of the data structure, you need to **reserve** space appropriately.
4. `vector<vector<shared_ptr<ServiceNode>>> findNegativeValueSubgraphs() const` which analyses the target `ServiceValueNetwork` object and returns any subgraph which exerts only negative value on the overall network. NOTE: a vector (the "external" **`return`** one in the signature) shall be used to return all the subgraphs, with each individual "internal" vector consisting of one distinct subgraph. In particular, you can implement a technique similar to Depth-First Search (DFS) described in Algorithm 1.

Finally, provide and discuss:

5. the **worst-case** complexity of all the methods you have implemented;
6. at least one opportunity to improve the `findNegativeValueSubgraphs()` e.g., by using alternative data structures or algorithmic processing.



```
Enter the number of nodes: 5
Enter the number of relationships: 4
Enter name and value for service 1: SteelFactory 1
Enter name and value for service 2: Taconite -3
Enter name and value for service 3: Hematite -7
Enter name and value for service 4: Coal -10
Enter name and value for service 5: Steel +1
Enter source and destination indices for relationship 1 (1 to 5): 1 2
Enter source and destination indices for relationship 2 (1 to 5): 1 3
Enter source and destination indices for relationship 3 (1 to 5): 1 4
Enter source and destination indices for relationship 4 (1 to 5): 1 5
```

Figure 3: Example of user prompts and inputs for question 3.

Algorithm 1 `findNegativeValueSubgraphs`

```
for each node $n$ do
    if value($n$) $< 0$ and $n$ not labeled as visited then
        initialize subgraph $G = \{\}$
        initialize stack $S = \{n\}$
        while $S$ holds elements do
            take node $c$ out of $S$
            if $c$ not labeled as visited then
                label $c$ as visited
                add $c$ to $G$
                for each $cn$ neighbor of $c$ with value($cn$) $< 0$
                  and not labeled as visited do
                    add $cn$ to $S$
        if $G$ holds elements then
            add $G$ to the list of found subgraphs
```

Solution 1

1. The method operates a simple addition operation of a node into the network. Specifically:

```
// Function to add a service node to the network
void addService(const std::string& service, int value) {
    auto newNode = std::make_shared<ServiceNode>(service, value); // Create a new node
    nodes.push_back(newNode); // Add it to the vector of nodes
}
```

2. The method simply adds an element within a network graph using a neighbors list for each node; as suggested, the `addNeighbor(const std::shared_ptr<ServiceNode>& neighbor)` method is used. Specifically:

```
// Updated method to add a relationship
void addRelationship(const std::shared_ptr<ServiceNode>& src, const std::shared_ptr<ServiceNode
  >& dest) {
    if (src && dest) {
        src->addNeighbor(dest);
    }
}
```

3. The complexity of the constructor comes from the user prompts to ask for the correct inputs. The rest simply reflects I/O operations. Specifically:

```
ServiceValueNetwork(int numberOfNodes, int numberOfRelationships) {
    nodes.reserve(numberOfNodes);  // Reserve space for nodes to avoid reallocations

    // Prompt the user to enter node data
    std::string serviceName;
    int serviceValue;
    for (int i = 0; i < numberOfNodes; ++i) {
        std::cout << "Enter name and value for service " << (i + 1) << ": ";
        std::cin >> serviceName >> serviceValue;
        addService(serviceName, serviceValue);
    }

    // Prompt the user to enter relationships
    int srcIndex, destIndex;
    for (int i = 0; i < numberOfRelationships; ++i) {
        std::cout << "Enter source and destination indices for relationship " << (i + 1) << " (1 to " <<
  numberOfNodes << "): ";
        std::cin >> srcIndex >> destIndex;
        if (srcIndex > 0 && srcIndex <= numberOfNodes && destIndex > 0 && destIndex <=
  numberOfNodes) {
            addRelationship(nodes[srcIndex - 1], nodes[destIndex - 1]);
        } else {
            std::cerr << "Error: Relationship indices out of range\n";
        }
    }
}
```

4. We implement the function for finding subgraphs of interconnected nodes with negative values without using recursion by using the algorithm described in the text. This algorithm avoids the potential stack overflow risk associated with deep recursion in large graphs and closely resembles the iterative DFS algorithm. In this solution, we use the `std::stack` container to represent the stack, but we also could rely on alternatives such as `std::deque`. Specifically:

```
std::vector<std::vector<std::shared_ptr<ServiceNode>>> ServiceValueNetwork::
    findNegativeValueSubgraphs()
      const {
    std::unordered_set<std::shared_ptr<ServiceNode>> visited;
    std::vector<std::vector<std::shared_ptr<ServiceNode>>> subgraphs;

    for (const auto& node : nodes) {
        if (node->getValue() < 0 && visited.find(node) == visited.end()) {
            std::vector<std::shared_ptr<ServiceNode>> subgraph;
            std::stack<std::shared_ptr<ServiceNode>> stack;

            stack.push(node);
            while (!stack.empty()) {
                auto currentNode = stack.top();
                stack.pop();

                if (visited.insert(currentNode).second) {
                    subgraph.push_back(currentNode);
                    for (const auto& neighbor : currentNode->getNeighbors()) {
                        if (neighbor->getValue() < 0 && visited.find(neighbor) == visited.end()) {
                            stack.push(neighbor);
                        }
                    }
                }
            }

            if (!subgraph.empty()) {
                subgraphs.push_back(subgraph);
            }
        }
    }

    return subgraphs;
}
```

5. The `addService` method exerts a worst-case computational complexity of $O(N)$, where N is the number of nodes copied. In `addRelationship`, we have a $O(1)$ complexity: the method is based on `addNeighbor`, which essentially performs a `push_back` in the `neighbors` list. As for the `findNegativeValueSubgraphs` method, the described DFS approach visits each node and each edge in the graph at most once. The traversal visits N nodes and, for each node, it processes all its outgoing edges, summing up to E edges in total. Therefore, in the worst case, where all nodes are negatively valued and interconnected in some way, the complexity is still linear with respect to the number of vertices and edges: $O(EN)$.

6. There are many ways to improve on the most simplistic implementation provided here. Any one of the options below works equally well (with varying degrees of improvement, of course). Specifically:

    - parallelization change - The method currently processes the graph sequentially, which can be inefficient on modern hardware where parallel processing capabilities are available. Therefore one solution is to offer a parallel version of DFS if the graph is large and the subgraphs are reasonably independent; the complexity of this case however is evaluated with means which are themselves beyond the scope of this course.
    - algorithmical change - The algorithm might traverse through nodes and edges that do not contribute to negative subgraphs. Therefore, one may introduce a marking system or a way to skip over already identified subgraphs or nodes that cannot be part of a negative subgraph. This involves more sophisticated tracking of graph connectivity and component marking but can save computational time by avoiding unnecessary work.
    - data-structural change - Nodes are checked linearly for negative values and connectivity. Therefore a valuable improvement could be to preprocess the graph with the intent of categorizing nodes and pre-build subgraphs dynamically as nodes are added or removed, maintaining an ongoing state of potential subgraphs.

Exercise 2 (14 points)

The process of converting a color photo to a black-and-white photo is called greyscaling. After greyscaling, it is useful to compute the gradient of the pixels of the image to detect regions of an image that look like edges.

You are required to implement a parallel program to greyscale a picture and to compute its gradient. Given a photo $\mathbf{p}$ with $N_r \times N_c$ pixels, where N_r is the number of rows and N_c the number of columns, one can compute its greyscaled values of each pixel $\mathbf{g}(i,j)$ as follows:

$$\mathbf{g}(i,j) = (0.299\,\mathbf{p}(i,j).r + 0.587\,\mathbf{p}(i,j).g + 0.114\,\mathbf{p}(i,j).b)/255, \tag{1}$$

with $i = 1,\ldots,N_r$ and $j = 1,\ldots,N_c$, and where $\mathbf{p}(i,j).r$ is the red component of the pixel in the i,j position, $\mathbf{p}(i,j).g$ is the green component and $\mathbf{p}(i,j).b$ is the blue component.
Next, we want to implement an algorithm to compute the gradient.
We consider the following strategy:

- Employ only two cores.
- Assume the number of rows to be a multiple of two.
- Split the picture along the rows, as in Figure 4.

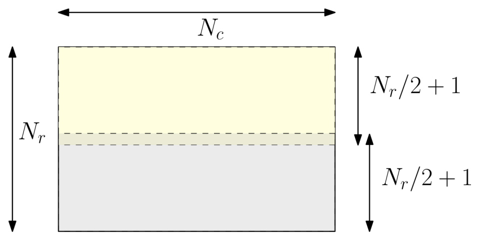


Figure 4: Sketch of the domain decomposition of the picture. Notice the overlapping region.

From the greyscaled image, compute the gradient with the following formula, only for the internal pixels. Hence, for $i = 2,\ldots,N_r-1,\ j = 2,\ldots,N_c-1$:

$$\mathbf{v}(i,j) = ((\mathbf{g}(i+1,j) - \mathbf{g}(i-1,j))^2 + (\mathbf{g}(i,j+1) - \mathbf{g}(i,j-1))^2)^{\frac{1}{2}}. \tag{2}$$

Notice that in a parallel implementation for computing the gradient, **each rank must store a copy of a common row**. For a code running with two ranks, we consider a decomposition as the one proposed in Figure 4. We split the photo into horizontal strips and require that a strip be shared between the two ranks. The photo is stored in the `struct Photo` with the three channels `r`, `g`, `b` being `std::vector<double>`. You are also provided a method `resizePhoto` to resize the vectors accordingly. Finally, reconstruct the gradient and the greyscaled picture on rank zero.

You have to complete the **main.cpp** file implementing the program described above. The initial part of the implementation is provided below:

```
#include <iostream>
#include <mpi.h>
#include <vector>
#include <cmath>
#include <fstream>

struct Photo
{
  std::vector<double> r;
  std::vector<double> g;
  std::vector<double> b;

  void resizePhoto(unsigned int dim){
    r.resize(dim);
    g.resize(dim);
    b.resize(dim);
  }
```

```
};

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if ( size != 2 )
  {
    if (rank == 0)
    {
      std::cout << "This program requires exactly 2 MPI processes" << std::endl;
    }
    MPI_Finalize();
    return 1;
  }

  // assume two domains
  Photo photo;
  unsigned int row, col;
  if (rank == 0)
  {
    std::ifstream inputFile("input.txt");
    inputFile >> row >> col;

    photo.resizePhoto(row*col);
    for (int i=0; i<row; ++i)
    {
      for (int j=0; j<col; ++j)
      {
        inputFile >> photo.r[i*col+j] >> photo.g[i*col + j] >> photo.b[i*col + j];
      }
    }
  }

  // YOUR CODE GOES HERE
  // declare local vectors

  if (rank == 0){

    // YOUR CODE GOES HERE
    // send the photo

  }
  else
  {
    // YOUR CODE GOES HERE
    // receive the photo
  }

  // YOUR CODE GOES HERE
  // compute the greyscale

  // compute the gradient

  // collect the results on rank zero
  MPI_Finalize();
}
```

Solution 2

An example solution is reported below. The difficulty is in computing the gradient. As suggested by the text, we communicate to the rank one a block of size $(N_r/2+1) \times N_c$, where the additional row is employed for the gradient computations. We compute the local grey-scaled photo and we use the data to compute the local gradient. Then we finally reconstruct both the grey-scaled photo and the gradient on rank zero.

```
#include <iostream>
#include <mpi.h>
#include <vector>
#include <cmath>
#include <fstream>

// We employ a simple struct to store the photo
// r g b are the vectors that store the red, green and blue values
// resizePhoto is a provided helper function to resize the vectors
struct Photo
{
  std::vector<double> r;
  std::vector<double> g;
  std::vector<double> b;

  void resizePhoto(unsigned int dim){
    r.resize(dim);
    g.resize(dim);
    b.resize(dim);
  }
};

int main(int argc, char *argv[])
{
  // initialize MPI
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // problem is simplified and solved with only two cores
  if ( size != 2 )
  {
    if (rank == 0)
    {
      std::cout << "This program requires exactly 2 MPI processes" << std::endl;
    }
    MPI_Finalize();
    return 1;
  }

  // declare photo
  Photo photo;
  unsigned int row, col;

  // read the photo at rank zero
  if (rank == 0)
  {
    // read the input file: gets row, cols and the photo
    std::ifstream inputFile("input.txt");
    inputFile >> row >> col;

    photo.resizePhoto(row*col);
    // reads the photo
    // the photo is a vector, but looped like a matrix
    for (int i=0; i<row; ++i)
    {
      for (int j=0; j<col; ++j)
      {
        inputFile >> photo.r[i*col+j] >> photo.g[i*col + j] >> photo.b[i*col + j];
      }
    }
  }

  // broadcast the dimensions of the photo to core one
  MPI_Bcast(&row, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
  MPI_Bcast(&col, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

  // assume two domains
  // Since we assumed even rows and only one core, we can have a simple split
  // that has overlap in the middle
  unsigned int local_row = row/2 + 1;

  // local vector to store the greyscale photo
  // the size of the vector is given by the col * local_row, where local_row collecs
  // also the overlapping row
  std::vector<double> localGrey(col*local_row);

  // local (to each core) photo for the rgb
  Photo localPhoto;
  localPhoto.resizePhoto(col*local_row);

  if (rank == 0){

    // send the photo to the other core
    // the core one needs a photo of size col*local_row, with the duplicate row.
    // The starting position from where to send the information is (row/2 - 1)*col
    // that becomes col * (local_row - 2)
    MPI_Send(&photo.r[col*(local_row-2)], col*local_row, MPI_DOUBLE, 1, 100, MPI_COMM_WORLD);
    MPI_Send(&photo.g[col*(local_row-2)], col*local_row, MPI_DOUBLE, 1, 100, MPI_COMM_WORLD)
      ;
    MPI_Send(&photo.b[col*(local_row-2)], col*local_row, MPI_DOUBLE, 1, 100, MPI_COMM_WORLD)
      ;

    // just make a simple copy for the first core
    for (unsigned int i=0; i<col*(local_row); ++i)
    {
      localPhoto.r[i] = photo.r[i];
      localPhoto.g[i] = photo.g[i];
      localPhoto.b[i] = photo.b[i];
    }

  }
  else
  {
    // recv the photo on core one: the receiving vector has to be at position 0
    MPI_Recv(&localPhoto.r[0], col*local_row, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
    MPI_Recv(&localPhoto.g[0], col*local_row, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
    MPI_Recv(&localPhoto.b[0], col*local_row, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  }

    // greyscale the photo
  for (unsigned int i=0; i< (col*local_row) ; ++i)
  {
    localGrey[i] = (0.299*localPhoto.r[i] + 0.587*localPhoto.g[i] + 0.114*localPhoto.b[i]) / 255.0;
  }

  // compute the gradient
  // the gradient dimension doesn't have the overlapping row.
  // Recall that the overlapping row is just a helper row to compute the gradient
  // but the gradient itself is computed only on the second block
  std::vector<double> localGrad(col*(local_row - 1), 0.0) ;

  for (unsigned int i=1; i<local_row-1; ++i)
  {
    for (unsigned int j=1; j < col - 1; ++j)
    {
      localGrad[i*col + j] = std::sqrt(
                                (localGrey[i*col + j + 1]   - localGrey[(i-1)*col + j - 1])*
                                (localGrey[i*col + j + 1]   - localGrey[(i-1)*col + j - 1])
                              + (localGrey[i*col + j + col] - localGrey[(i+1)*col + j - col])*
                                (localGrey[i*col + j + col] - localGrey[(i+1)*col + j - col])
                               );
    }
  }

  std::vector<double> grey;
  std::vector<double> gradient;

  // collect the results
  if (rank == 0){
    grey.resize(row*col);
    gradient.resize(row*col);

    for (std::size_t i=0; i < localGrad.size(); ++i)
    {
          grey[i] = localGrey[i];
      gradient[i] = localGrad[i];
    }

    // collect the result ignoring the overlapping row
    MPI_Recv(     &grey[col*(local_row - 1)], col*(local_row-1), MPI_DOUBLE,
                                    1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&gradient[col*(local_row - 1)], col*(local_row-1), MPI_DOUBLE,
                                    1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  }
  else{
    // send the result ignoring the overlapping row
    MPI_Send(&localGrey[col], col*(local_row-1), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Send(&localGrad[0], col*(local_row-1), MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
```

```
160   }
161
162
163   // this is just a nice way to output the results
164   if (rank == 0){
165     std::ofstream outFile("greyscale.out");
166     for (const auto & elem : grey)
167     {
168       outFile << elem << "\n";
169     }
170     outFile.close();
171   }
172
173   if (rank == 0){
174     std::cout << std::endl;
175     auto minGrad = std::min_element(gradient.begin(), gradient.end());
176     auto maxGrad = std::max_element(gradient.begin(), gradient.end());
177
178     std::ofstream outFile("example/gradient.out");
179     for (const auto & elem : gradient)
180     {
181       outFile << (elem - *minGrad) / (*maxGrad - *minGrad) << "\n";
182     }
183     outFile.close();
184   }
185
186   MPI_Finalize();
187 }
```

Note that at lines 66/67, we also could have used `Send` instead of `Bcast` given the limited number of ranks.

Exercise 3 (4 points)

The provided code simulates the stock of an electronics shop by implementing the `Item`, `Device`, `Smartphone`, and `Computer` classes. The `main` function serves as a practical demonstration of the system functionalities. Items can be sold, added to stock, and returned by users. In addition, each user can decide to add a warranty (Italian: *garanzia*) to the purchased product lasting an integer number of years of their choice.

After carefully reading the code, you have to answer the following questions. Please make sure to **highlight 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 quantity of smartphones `phoneA` in stock, printed at line 21 of the `main`?
2. What is the quantity of smartphones `phoneB` in stock, printed at line 22?
3. What is the number of computers costing less than $1000, printed on line 40?
4. What is the value of `cash_register` at line 41?

Provided source code:

- **Item.hpp**

```
#ifndef ELECTRONICS_ITEM_H
#define ELECTRONICS_ITEM_H

#include <iostream>

class Item {
protected:
    double price = 0.0;
    double discount = 0.0;
    unsigned quantity = 0;
    std::string brand = "";
public:
    Item() = default;
    Item(double price, double discount, unsigned quantity, const std::string &brand): price(price),
      discount(discount), quantity(quantity), brand(brand) {}
    unsigned get_quantity() const;
    double sell();
    double get_actual_price() const;
    virtual void add_to_stock(unsigned number_of_items);
    virtual double return_item(double original_price) = 0;
};

#endif //ELECTRONICS_ITEM_H
```

- **Item.cpp**

```
#include "Item.h"

double Item::sell() {
    if (quantity > 0) {
        quantity -= 1;
        return get_actual_price();
    }
    std::cout << "Item not found in the stock";
    return 0.0;
}

double Item::get_actual_price() const {
    return (1-discount)*price;
}

void Item::add_to_stock(unsigned number_of_items) {
    quantity += number_of_items;
}

unsigned Item::get_quantity() const {
    return quantity;
}
```

- **Device.hpp**

```
#ifndef ELECTRONICS_DEVICE_H
#define ELECTRONICS_DEVICE_H

#include "Item.h"

class Device: public Item{
protected:
    double storage = 0.0;
    double memory = 0.0;
public:
    Device() = default;
    Device(double price, double discount, unsigned quantity, const std::string &brand, double storage,
      double memory):
            Item(price, discount, quantity, brand), storage(storage), memory(memory) {}
    virtual double buy_warranty(unsigned years) const = 0;
    double return_item(double original_price) override;
};

#endif //ELECTRONICS_DEVICE_H
```

- **Device.cpp**

```
#include "Device.h"

double Device::return_item(double original_price) {
    quantity += 1;
    return original_price;
}
```

- **Smartphone.hpp**

```
#ifndef ELECTRONICS_SMARTPHONE_H
#define ELECTRONICS_SMARTPHONE_H

#include <iostream>
#include "Device.h"

class Smartphone: public Device {
private:
    std::string connectivity = "";
    const unsigned max_items = 15;
public:
    Smartphone() = default;
    Smartphone(double price, double discount, unsigned quantity, const std::string &brand, double
      storage, const std::string &OS, double memory, unsigned core_number, const std::string &
      connectivity):
            Device(price, discount, quantity, brand, storage, memory), connectivity(connectivity) {}
    double buy_warranty(unsigned years) const override;
    void add_to_stock(unsigned number_of_items) override;
};

#endif //ELECTRONICS_SMARTPHONE_H
```

- **Smartphone.cpp**

```
#include "Smartphone.h"

double Smartphone::buy_warranty(unsigned years) const {
    if (years <= 2)
        return 50*years;
    return 100 + 25*years;
}

void Smartphone::add_to_stock(unsigned number_of_items) {
    quantity += number_of_items;
    if (quantity > max_items)
        quantity = max_items;
}
```

- **Computer.hpp**

```
#ifndef ELECTRONICS_COMPUTER_H
#define ELECTRONICS_COMPUTER_H

#include <iostream>
```

```
#include "Device.h"

class Computer: public Device{
private:
    double GPU_memory = 0.0;
public:
    Computer() = default;
    Computer(double price, double discount, const std::string &brand, double storage, const std::string
      &OS, double memory, unsigned core_number, double GPU_memory):
            Device(price, discount, 10, brand, storage, memory), GPU_memory(GPU_memory) {}
    Computer(double price, double discount, unsigned quantity, const std::string &brand, double
      storage, const std::string &OS, double memory, unsigned core_number, double GPU_memory):
            Device(price, discount, quantity, brand, storage, memory), GPU_memory(GPU_memory) {}
    double return_item(double original_price) override;
    double buy_warranty(unsigned years) const override;
};

#endif //ELECTRONICS_COMPUTER_H
```

- **Computer.cpp**

```
#include "Computer.h"

double Computer::return_item(double original_price) {
    quantity += 1;
    return 0.5*original_price;
}

double Computer::buy_warranty(unsigned years) const {
    return 0.1*price*years;
}
```

- **main.cpp**

```
#include <iostream>
#include "Smartphone.h"
#include "Computer.h"
#include <map>

int main() {
    std::map<std::string, Smartphone> smartphones;
    std::map<std::string, Computer> computers;
    double cash_register = 0.0;

    smartphones.insert({"phoneA", Smartphone(120.00, 0.25, 8, "Apple", 8.0, "iOS", 1, 2, "5G")});
    smartphones.insert({"phoneB", Smartphone(400, 0, 10, "Samsung", 256, "Android", 64, 2, "5G")});
    smartphones.insert({"phoneA", Smartphone(250.00, 0.2, 5, "Apple", 8.0, "iOS", 1, 2, "5G")});

    smartphones["phoneB"].add_to_stock(10);
    cash_register += smartphones["phoneA"].sell();
    cash_register += smartphones["phoneB"].sell() + smartphones["phoneB"].buy_warranty(4);
    cash_register += smartphones["phoneA"].sell() + smartphones["phoneA"].buy_warranty(2);
    cash_register -= smartphones["phoneA"].return_item(200.00);

    std::cout << smartphones["phoneA"].get_quantity() << std::endl;
    std::cout << smartphones["phoneB"].get_quantity() << std::endl;

    computers.insert({"pc1", Computer(800.00, 0.25, 20, "Acer", 512.0, "Windows", 16, 7, 4)});
    computers.insert({"pc2", Computer(1600.00, 0.0, 10, "Apple", 2048.0, "macOS", 32, 12, 8)});
    computers.insert({"pc3", Computer(1250.00, 0.2, "Asus", 1024.0, "Windows", 12, 5, 0)});

    cash_register += computers["pc1"].sell() + computers["pc1"].buy_warranty(2);
    cash_register += computers["pc2"].sell();
    cash_register -= computers["pc1"].return_item(1000.00);
    cash_register -= computers["pc3"].return_item(1080.00);
    computers["pc1"].add_to_stock(5);

    unsigned count = 0;
    for (const auto it: computers){
        if (it.second.get_actual_price() <= 1000.00)
            count += it.second.get_quantity();
    }

    std::cout << count << std::endl;
    std::cout << cash_register << "$" << std::endl;

    return 0;
}
```

Solution 3

1. **7**.
   The element with the key `phoneA` is initialized with eight units and is not overwritten when attempting to insert an element with the same key. Two units are sold, and then one unit is returned.

2. **14**.
   The element with the key `phoneB` is initialized with ten units. The function `add_to_stock` adds five more smartphones of this type. Then, one unit is sold.

3. **36**.
   Only `pc1` and `pc3` have a cost less than or equal to $1000. For `pc1`, the element is initialized with twenty units; one is sold, and one is returned. Then, five more units are added to the stock, making a total of 25 units. For `pc3`, the element is initialized with ten units, and one is returned, making a total of 11 units.

4. **2000**.
   According to the code, the value of `cash_register` is updated as follows: 90, 690, 880, 680, 1440, 3040, 2540, and finally 2000.