Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 31-8-2021

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

Exam duration: 2 hours

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 (14 points)

You have to develop the `TimeSeries` class to manage efficiently time series, i.e., **series of data points indexed in time order**. In particular, your goal is to store sequences of **float** values and to optimize the **worst case complexity** of all methods.

The time is denoted by a two level granularity, i.e., by specifying a date and a time stamp. Both are represented as strings in the format *"yyyy/mm/dd"* and *"hh:mm:ss"*, respectively. Note that in this way, by relying on the order relation of strings, the order relations for dates and times are preserved (time goes from "00:00:00" to "23:59:59").

You have to:

1. provide the declaration of the `TimeSeries` class (you can focus only on the data structures and any additional methods you possibly will introduce but you can omit from the declaration the methods listed below).

2. Implement the method

    void add_registration(**const** string & date, **const** string & time, **float** value);

    which adds a value to the time series.

3. **float** get_registration(**const** string & date, **const** string & time) **const**;

    which returns the time series value at `date` and `time` or NaN in case the value was not previously added in the time series.

4. provide alternatively the implementation of

    float get_registration_by_date(**const** string & date) **const**;

    or

    float get_registration_by_time(**const** string & time) **const**;

    which compute the average across all the values stored at a given date or time stamp (note that, even if it is requested to implement one method, **the data structures you choose have to optimize the worst case complexity of both**).

5. vector<**float**> get_registration_range(**const** string & date1, **const** string & time1, **const** string & date2) **const**;

    which returns the set of values stored starting from date1 and time1 up to date2. date2 data should not be included in the result. If date2<date1 or date1 is not included in the time series, the vector returned as result is empty.

6. `TimeSeries intersect_time_series(const TimeSeries & ts) const;`

   which returns the intersection of the time series with the one specified as parameter. The values of the `TimeSeries` returned as result are obtained as mean of the corresponding values of the two initial time series. For example, if the two intial time series stores the registrations ("2021/08/31", "11:30:00", 1) and ("2021/08/31", "11:30:00", 2) the registration ("2021/08/31", "11:30:00", 1.5) is stored in the result.
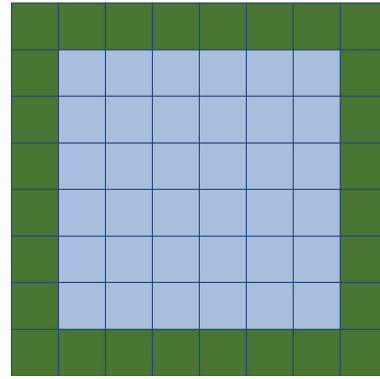
7. Finally, provide the **worst case complexity** of all the methods you have implemented (with the exception of the intersect method).

Note that you can include any additional helper methods if needed, but in that case you have to provide also their implementation and complexity.

Exercise 2 (11 points)
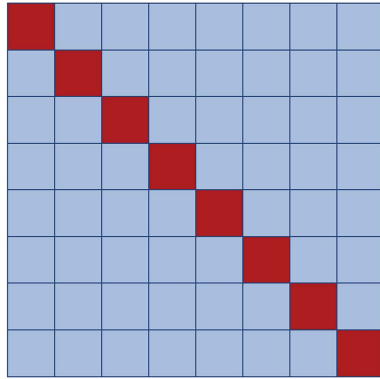
$$\text{trace} = \sum_{i=1}^{n} A_{ii} \qquad \text{crown} = \sum_{i=1}^{n} (A_{1i} + A_{ni} + A_{i1} + A_{in})$$

$$\text{fcrown} = \text{trace} \cdot \text{crown}$$

Figure 1: *fcrown* property computation

You have to implement a **parallel function** with the following prototype:

`double fcrown_property (const std::string& filename);`

It computes the *fcrown property* of a **squared matrix A**, which should be **read from the file** whose name is passed as parameter and represented through the class `la::dense_matrix` (header file reported below).

The *fcrown property* of matrix A is defined as:

$$\text{fcrown}(A) = \text{trace}(A) \cdot \text{crown}(A),$$

where

$$\text{trace}(A) = \sum_{i=1}^{n} A_{ii}$$

and

$$\text{crown}(A) = \sum_{i=1}^{n} (A_{1i} + A_{ni} + A_{i1} + A_{in})$$

as reported in Figure 1. Note that, when computing crown(A), **the corner elements are considered twice**. For instance, element A_{11} is included both when summing the elements on the first row and when summing the elements in the first column (other than when computing the trace of the matrix).

For the computation, you can assume that **the number of rows in the matrix is multiple of the number of available cores**.

The header file of the `dense_matrix` class is provided below.

*** dense_matrix.hpp

```cpp
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, m_columns;
    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;
  };

  dense_matrix
```

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

void
swap (dense_matrix &, dense_matrix &);
}
```

Exercise 3 (8 points)

The code provided below models the interactions between a mobile phone and the tariffs applied by its carrier on operations like sending messages and making calls.

Each `Carrier` sets a cost per sms sent, as well as a cost per minute spent in a call. Such costs can be retrieved through dedicated methods, while the `updatePrices()` method allows a carrier to change the costs applied to users.

The class `Phone` contains the information relative to a single phone, that include a list of contacts and an available credit together with a pointer to the current carrier. The method `textToContact()` will send a message to a contact and charge the user for the message only if the phone has enough credit and the contact is registered in the phone. The method `callContact()` will call a contact for the requested duration, unless the available credit does not allow for a call of such length. The method `textAllContacts()` will attempt to broadcast the same text to all registered contacts, and finally the method `changeCarrier()` allows a phone owner to change carrier to the phone. The complete implementation of both classes is reported below.

After carefully reading the code, you have to answer the following questions **(in the provided web form if you are attending the online version of the exam). Notice that each answer requires you to type a single number!** Moreover, **it is mandatory for you to develop your solution motivating the results you achieved on the sheets** (that will be possibly uploaded). **Only providing the final answers is not enough for obtaining points in this section.**

1. What is the credit of the phone after the exection of `textToContact()` at line 10?

2. What is the value stored in `callTime` after the execution of `callContact()` at line 11?

3. What is the value stored in `messageSent` after the execution of `textToContact()` at line 14? (1 for true, 0 for false)

4. Depending on the last answer, either the instruction `callContact()` at line 18 or `textAllContacts()` is executed. What is the value returned by the executed instruction?

Provided source code:

- **Product.h**

  ```
  #include <string>

  class Carrier {
  private:
      unsigned smsCost;
      unsigned callCostPerMinute;

  public:
      Carrier(unsigned a, unsigned b);

      unsigned billSms();
      unsigned billPhoneCall(unsigned minutes);

      void updatePrices(unsigned sms, unsigned call = 0);

      void sendMessage(std::string msg);

  };
  ```

- **Product.cpp**

```
#include "Carrier.h"

Carrier::Carrier(unsigned a, unsigned b) : smsCost(a), callCostPerMinute(b) {}

unsigned Carrier::billSms() {
  return smsCost;
}

unsigned Carrier::billPhoneCall(unsigned minutes) {
  return callCostPerMinute * minutes;
}

void Carrier::updatePrices(unsigned sms, unsigned call) {
  smsCost = sms;
  callCostPerMinute = call ? call : callCostPerMinute;
}

void Carrier::sendMessage(std::string msg) {
  /* Implementation is irrelevant to the exercise */
}
```

- **Shop.h**

```
#include <memory>
#include <unordered_set>
#include "Carrier.h"

class Phone {
private:
    std::unordered_set<std::string> contacts;
    unsigned credit;
    std::shared_ptr<Carrier> carrier;

public:
    Phone(std::unordered_set<std::string> c, unsigned cr,
        std::shared_ptr<Carrier> ca);

    bool textToContact(std::string contact, std::string content);

    unsigned callContact(std::string contact, unsigned mins);

    unsigned textAllContacts(std::string msg);

    void changeCarrier(std::shared_ptr<Carrier> c);
};
```

- **Shop.cpp**

```
#include "Phone.h"

Phone::Phone(std::unordered_set<std::string> c, unsigned cr,
          std::shared_ptr<Carrier> ca) : contacts(c), credit(cr),
                              carrier(ca) {}

bool Phone::textToContact(std::string contact, std::string content) {
  if (contacts.count(contact) && credit >= carrier->billSms()) {
    credit -= carrier->billSms();
    carrier->sendMessage(content);
    return true;
```

```cpp
      }
      return false;
    }

    unsigned int Phone::callContact(std::string contact, unsigned int mins) {
      unsigned m = 0;
      if (contacts.count(contact)) {
        while (m < mins && credit > carrier->billPhoneCall(m + 1)) {
          ++m;
        }
        credit -= carrier->billPhoneCall(m);
      }
      return m;
    }

    unsigned int Phone::textAllContacts(std::string msg) {
      unsigned ret = 0;
      bool canSend = true;
      for (std::string c : contacts) {
        if (canSend) {
          canSend = textToContact(c, msg);
        }
        ret = ret + canSend;
      }
      return ret;
    }

    void Phone::changeCarrier(std::shared_ptr<Carrier> c) {
      carrier = c;
    }
```

- **main.cpp**

```cpp
1   #include <iostream>
2   #include "Phone.h"
3
4   int main() {
5
6     std::shared_ptr<Carrier> blueCarrier = std::make_shared<Carrier>(3, 4);
7     std::shared_ptr<Carrier> redCarrier = std::make_shared<Carrier>(5, 5);
8     Phone phone({"Alice", "Betty", "Charles", "Mom"}, 75, blueCarrier);
9
10    phone.textToContact("Robert", "Hi rob, how are you?");
11    unsigned callTime = phone.callContact("Mom", 12);
12    std::cout << "First call time is " << callTime << std::endl;
13    blueCarrier->updatePrices(10);
14    bool messageSent = phone.textToContact("Betty", "Please call me at 9:30 pm");
15    phone.changeCarrier(redCarrier);
16    if (!messageSent) {
17      std::cout << "Last call lasted ";
18      std::cout << phone.callContact("Alice", 5);
19      std::cout << " minutes" << std::endl;
20    } else {
21      std::cout << "Text sent to ";
22      std::cout << phone.textAllContacts("You are invited to my party!!");
23      std::cout << " contacts" << std::endl;
24    }
25    return 0;
26  }
```