Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 14-06-2022

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

Exam duration: 2 hours and 15 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 (14 points)

A tensor is a data container which can store data in N dimensions. You have to develop a class implementing tensors up to $N = 3$ dimensions for **double**s. You have three main goals: i) your implementation should optimize the **worst case complexity**, ii) you need to support **random access**, iii) your data structure will be **sparse**.

Considering that any index starts from 0, you have to:

1. implement the constructor which receives as input parameter the dimensions of your `Tensor` class
2. provide the definition of the method:

   ```
   void set(double value, unsigned i, int j=-1, int k= -1)
   ```

   k has default value -1 to denote the scenarios where the tensor has one or two dimensions, while j default value -1 characterizes one dimensional tensors. This method will add (for an unseen index) or update (in case the index already exists) values in your tensor objects.

3. implement the method:

   ```
   double get(unsigned i, int j=-1, int k= -1) const
   ```

   k has default value -1 to denote the scenarios where the tensor has one or two dimensions, while j default value -1 characterizes one dimensional tensors. In other words, if t1 is a 1-dim tensor, t2 is a 2-dim tensor and t3 is a 3-dimensional tensor, the user of your class can write the following code raising in some scenarios error conditions:

   ```
   t1.set(1,0); //OK t1[0]=1
   t2.set(-2,0,1); //OK t2[0][1]=2
   t2.set(1,0,0); //OK t2[0][0]=1
   t2.set(2,0,0); //OK t2[0][0]=2
   t3.set(3,2,2,2); //OK t3[2][2][2]=3
   std::cout << t1.get(0) << std::endl; //OK
   std::cout << t2.get(0) << std::endl; // Index not appropriate
   std::cout << t2.get(0,0) << std::endl; // OK
   std::cout << t3.get(2,2,2) << std::endl; // OK
   std::cout << t3.get(3,3,3) << std::endl; // Index not appropriate
   std::cout << t1.get(1) << std::endl; // Index not appropriate
   std::cout << t2.get(1,0) << std::endl; // Index not appropriate
   ```

In your implementation **you have to cope with error conditions** and check if indexes are appropriate. In case of error `get` will return a NaN while `set` will not have effect in your data structure.

Moreover, provide the implementation and complete the declarations of the following methods **(you have to consider also error conditions and add any missing const qualifiers)**:

4. `Tensor operator*(double alpha)`: the product of the tensor with a double
5. `double norm(const std::string& norm_type))`: the method computes entry-wise tensor norms (i.e., each tensor element is treated as a vector element). `norm_type` can be "inf", "euclidean", "2" (same as euclidean), "fro" (i.e., Frobenius norm). Note that "fro" can be applied only if the tensor is a bi-dimensional matrix.
6. `void reshape(.... new_shape)`: **discuss (providing high level ideas, no code is required)** how you would implement this method that can change the tensor size (for example, a two-dimensional tensor t2 can be transformed into a one-dimensional tensor as you do in matlab through t2(:)).

Additionally, provide:

7. the **worst complexity** of all the methods you have implemented.

Solution 1

In order to optimize the worst case complexity and implement a sparse data structure, the best option is to rely on a `std::map`. For what concerns the index choice, since the dimensions of the tensors are provided with the constructor, the best choice is to think to the *vectorised* representation of the vector as we implemented during our course in the `dense_matrix` class and to map the indexes to an unsigned value (the mapping is straightforward for 1-dim and 2-dim tensors while for three dimensional tensors you can convert the access to the element t3[i][j][k] to the element of index $i * 2\_dim\_size + j * 3\_dim\_size + k$). Below we provide an alternative implementation where the index of the `std:map` is a `std::vector`. Remind that vectors already provide an `operator<` and so can be used as `std:map` keys. The main advantage of the former implementation is that the `reshape` method would come almost for free, i.e., you would just need to change the internal tensor dimension representation to cope with error conditions with a $O(1)$ complexity. Vice versa, the latter implementation based on vectors as keys would require to scan all elements of the tensor and to generate a new tensor with the index representation with complexity $O(Mlog(M))$ where M is the number of elements in the tensor.

The declaration of the class is reported below:

```
#ifndef TENSORS_TENSOR_H
#define TENSORS_TENSOR_H

#include <vector>
#include <map>
#include <string>

class Tensor {
public:

    typedef std::vector<unsigned> index_type;

private:
    std::vector<unsigned> dimensions;
    std::map<index_type, double> values;

    double infinity_norm() const;

    double euclidean_norm() const;

    bool check_indexes(unsigned i, int j, int k) const;

    index_type compute_index(unsigned i, int j, int k) const;

public:
    Tensor(const index_type &dimensions);

    void reshape(index_type new_shape);

    double get(unsigned i, int j=-1, int k= -1) const;

    void set(double value, unsigned i, int j=-1, int k= -1);

    Tensor operator*(double alpha) const;

    /* compute tensor norm.
     * norm_type can be "inf", "euclidean", "2" (same as euclidean), "fro"
     * Note that "fro" can be applied only if the tensor is a bi-dimensional matrix
     * */
    double norm(const std::string& norm_type) const;

    void print() const;

};


#endif //TENSORS_TENSOR_H
```

The methods implementations, accordingly are as follows:

```
Tensor::Tensor(const index_type& dim) : dimensions(dim) {
    if (dim.size()==0)
        std::cerr << "0 dimensional Tensor created. FATAL ERROR" << std::endl;

}

double Tensor::infinity_norm() const {
    double norm = -1;

    for (auto cit = values.cbegin(); cit != values.cend();++cit)
        if (abs(cit->second) > norm)
            norm = abs(cit->second);

    return norm;
}

double Tensor::euclidean_norm() const {
    double norm = 0;

    for (auto cit = values.cbegin(); cit != values.cend();++cit)
        norm += cit->second * cit->second;

    return sqrt(norm);
}


double Tensor::get(unsigned int i, int j, int k) const {
    if (!check_indexes(i, j, k)) {
        std::cerr << "Index not appropriate" << std::endl;
        return std::numeric_limits<double>::quiet_NaN();
    }
    else{
        index_type indexes = compute_index(i, j, k);
        if (values.find(indexes)!=values.cend())
            return values.at(indexes);
        else{
            std::cerr << "Index not appropriate" << std::endl;
            return std::numeric_limits<double>::quiet_NaN();
        }


    }

}

void Tensor::set(double value, unsigned int i, int j, int k) {
    if (!check_indexes(i, j, k)) {
        std::cerr << "Index not appropriate" << std::endl;
    }
    else{
        index_type indexes = compute_index(i, j, k);
        values[indexes] = value;

    }
}

Tensor Tensor::operator*(double alpha) const {
    Tensor result(dimensions);
    for (auto cit = values.cbegin(); cit != values.cend();++cit)
        result.values[cit->first]= alpha* cit->second;

    return result;
}
    /* compute tensor norm.
     * norm_type can be "inf", "euclidean", "2" (same as euclidean), "fro"
     * Note that "fro" can be applied only if the tensor is a bi-dimensional matrix
     * */
double Tensor::norm(const std::string& norm_type) const {

    if (norm_type == "inf")
        return infinity_norm();
    else if (norm_type == "euclidean" or norm_type == "2" )
        return euclidean_norm();
    else if (norm_type == "fro"){
          if(dimensions.size()==2)
              return euclidean_norm();
          else
              std::cerr << "Frobenius norm can be computed only for matrices" << std::endl;
    }
    else{
        std::cerr << "Unknown norm" << std::endl;
        return std::numeric_limits<double>::quiet_NaN();
    }

    return std::numeric_limits<double>::quiet_NaN();
}

bool Tensor::check_indexes(unsigned int i, int j, int k) const{

    if (dimensions.size()==0 or dimensions.size()>3)
        return false;

    if (i>= dimensions[0])
```

```
        return false;

    // first check if index is used and dimensions are not appropriate
    // then check if dimensions are such that dimensions[1] can be accessed and j is out of bound
    if ((j>=0 and dimensions.size()==1) or (dimensions.size()>1 and j>= dimensions[1]))
            return false;

    if ((k>=0 and dimensions.size()<=2) or (dimensions.size()>2 and k>= dimensions[2]))
        return false;

    // If we reach the code here the index is appropriate
    return true;


}

Tensor::index_type Tensor::compute_index(unsigned int i, int j, int k) const {
    index_type new_index(dimensions.size());
    new_index[0] = i;
    if (j>=0)
        new_index[1]=j;
    if (k>=0)
        new_index[2]=k;

    return new_index;
}
```

Given the data structure choice, the most important method is `check_indexes()` . The method first checks if the tensor has been created with a proper size (i.e., the dimensions is greater than zero and less or equal to three). If this control is successful, then the method checks if the index is used properly according to the tensor dimensions and is in the proper range. The complexity of this method is $O(1)$. The `compute_index` method implementation is then straightforward and it also brings $O(1)$ complexity. Vice versa, the `get` and `set` method, since they access a `std::map`, are characterized by a $log(M)$ complexity. Finally, the norms computation has complexity $O(M)$ while the scalar product has complexity $O(Mlog(M))$, due to the M inserts in the new tensor, each with complexity $O(log(M))$.

Exercise 2 (11 points)

Discrete-time random processes represent situations where a system evolves between multiple states, that are reached with a given probability. Assuming that the number of possible states is known and equal to n, the probability of moving from a state $X_t = i$ to a state $X_{t+1} = j$ is usually represented, for all i and j, through an element p_{ij} of the transition matrix $P \in \mathbb{R}^{n\times n}$. In other words, we define:

$$p_{ij} = \mathcal{P}\left(X_{t+1} = j | X_t = i\right). \tag{1}$$

Note that the elements on the diagonal represent the probability of remaining in the current state (i.e., for instance, p_{11} is the probability of remaining in state $X_{t+1} = X_t = 1$).

By definition, P is a probability matrix, i.e., the following conditions are satisfied:

$$p_{ij} \in [0,1] \quad \forall\, i,j = 1,\ldots n \tag{2}$$

$$\sum_{j=1}^{n} p_{ij} = 1 \quad \forall\, i = 1,\ldots n. \tag{3}$$

- **You have to implement** a **parallel function** to check if a given matrix P is a probability matrix, with the following prototype:

  ```
  bool check_probability_matrix (const la::dense_matrix& P);
  ```

It receives as parameter a matrix $P \in \mathbb{R}^{n\times n}$, represented as an object of the `la::dense_matrix` class, and it returns `true` if `P` is a probability matrix (`false` otherwise), i.e., if conditions (2) and (3) are satisfied.

You can assume that **the matrix `P` is known to all processes** and that the number of elements n **is a multiple of the number of available cores**.

The transition matrix P can be used to compute the probability of any trajectory T, defined as a sequence of states (i.e., $T = \{X_1, X_2, X_3, \dots\}$). In particular, under the assumption that the next state X_{t+1} depends only on the current state X_t, and not on all the previous history $X_1, \dots X_{t-1}$, this probability is computed as follows.

Let $T = \{2, 3, 1, 1\}$ be a sample trajectory, meaning that the system moves from state $X_1 = 2$ to state $X_2 = 3$, than to $X_3 = 1$, etc. The probability associated to this trajectory is:

$$P(T) = P(X_2 = 3|X_1 = 2) \cdot P(X_3 = 1|X_2 = 3) \cdot P(X_4 = 1|X_3 = 1) = p_{23} \cdot p_{31} \cdot p_{11}. \tag{4}$$

As an example, given the following transition matrix,

$$P = \begin{bmatrix} 0.1 & 0.2 & 0.7 \\ 0.3 & 0.3 & 0.4 \\ 1.0 & 0.0 & 0.0 \end{bmatrix}, \tag{5}$$

the probability of the trajectory $T = \{2, 3, 1, 1\}$ is:

$$P(T) = P(X_2 = 3|X_1 = 2) \cdot P(X_3 = 1|X_2 = 3) \cdot P(X_4 = 1|X_3 = 1) = p_{23} \cdot p_{31} \cdot p_{11} = 0.4 \cdot 1.0 \cdot 0.1 = 0.04.$$

- **You have to implement** another **parallel function** to compute the probabilities of multiple trajectories, with the following prototype:

  ```
  std::vector<double> compute_probabilities (const la::dense_matrix& P,
                     const std::vector<std::vector<unsigned>>& trajectories);
  ```

  It receives as parameter a probability matrix $P \in \mathbb{R}^{n\times n}$ and a vector of trajectories (each trajectory is represented as a vector of indices), and it returns a vector with the probabilities of all the given trajectories.

  You can assume that the matrix **`P` is known to all processes**, while the vector of **trajectories is know only to rank `0`. The number of trajectories is a multiple of the number of available cores**.

The declaration of the `dense_matrix` class is reported below:

```
#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, 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 &);
}

#endif // DENSE_MATRIX_HH
```

Solution 2

- A possible implementation for the check_probability_matrix function is reported in the following:

```
bool check_probability_matrix (const la::dense_matrix& P)
{
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
 8
 9    int valid = 1;
10
11    /*
12      P is a probability matrix if:
13        > P(i,j) \in [0,1] \forall i,j
14        > \sum_{j}P(i,j) = 1 \forall i
15    */
16
17    // loop over matrix rows
18    for (unsigned i = rank; i < P.rows() && valid == 1; i+=size)
19    {
20      double sum = 0.;
21
22      for (unsigned j = 0; j < P.columns() && valid == 1; ++j)
23      {
24        if (P(i,j) < 0. || P(i,j) > 1.)
25          valid = 0;
26        sum += P(i,j);
27      }
28
29      if (sum < 1.0 || sum > 1.0)
30        valid = 0;
31    }
32
33    MPI_Allreduce(MPI_IN_PLACE, &valid, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
34
35    return (valid > 0);
36  }
```

As you can notice, the cyclic partitioning schema is chosen, relying on the fact that the matrix is by assumption known to all processes. We check in line 24 that each element of P belongs to the interval $[0, 1]$, while we check in line 29 that the sum of all elements in a row is equal to 1.

Note that we decided to use an **int** to save the validity of the rows examined in each process, and then to accumulate the results by means of an MPI_Allreduce, exploiting the MPI_PROD operation. Since all variables are 1 if the matrix is a probability matrix, while at least one is 0 if a condition is violated, the product will be 1 or 0 accordingly.

As an alternative, the MPI_LAND operation, which performs the logical AND between variables, could be used to accumulate the results.

- A possible implementation for the compute_probabilities function is reported in the following:

```
38  std::vector<double> compute_probabilities (const la::dense_matrix& P,
39                       const std::vector<std::vector<unsigned>>& trajectories)
40  {
41    int rank, size;
42    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
43    MPI_Comm_size(MPI_COMM_WORLD, &size);
44
45    // get the total number of trajectories
46    unsigned n_trajectories = trajectories.size();
47    MPI_Bcast(&n_trajectories, 1, MPI_UNSIGNED, 0,MPI_COMM_WORLD);
48
49    // compute the local number of trajectories and initialize the vector
50    unsigned local_n_trajectories = n_trajectories / size;
51    std::vector<std::vector<unsigned>> local_trajectories(local_n_trajectories);
52
53    // send/recv trajectories
54    if (rank == 0)
```

```
  {
    for (unsigned local_j = 0; local_j < local_n_trajectories; ++local_j)
      local_trajectories[local_j] = trajectories[local_j];

    for (int r = 1; r < size; ++r)
    {
      for (unsigned local_j = 0; local_j < local_n_trajectories; ++local_j)
      {
        unsigned j = local_j + r * local_n_trajectories;

        // compute and send the number of states in the trajectory
        unsigned n_states = trajectories[j].size();
        MPI_Send(&n_states, 1, MPI_UNSIGNED, r, 0, MPI_COMM_WORLD);

        // send the trajectory
        MPI_Send(trajectories[j].data(), n_states, MPI_UNSIGNED, r, 0,
                 MPI_COMM_WORLD);
      }
    }
  }
  else
  {
    for (unsigned j = 0; j < local_n_trajectories; ++j)
    {
      // receive the number of states
      unsigned n_states;
      MPI_Recv(&n_states, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);

      // initialize the local trajectory and receive it
      local_trajectories[j].resize(n_states);
      MPI_Recv(local_trajectories[j].data(), n_states, MPI_UNSIGNED, 0, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
  }

  // compute probabilities
  std::vector<double> local_probabilities(local_n_trajectories, 1.0);
  for (unsigned j = 0; j < local_n_trajectories; ++j)
  {
    const std::vector<unsigned>& trajectory = local_trajectories[j];
    for (unsigned i = 1; i < trajectory.size(); ++i)
      local_probabilities[j] *= P(trajectory[i-1], trajectory[i]);
  }

  // aggregate local results
  std::vector<double> probabilities(n_trajectories);
  MPI_Allgather(local_probabilities.data(), local_n_trajectories, MPI_DOUBLE,
                probabilities.data(), local_n_trajectories, MPI_DOUBLE,
                MPI_COMM_WORLD);

  return probabilities;
}
```

Note that `MPI_Scatter` operation cannot be used to divide the `trajectories` vector between the different processes, since each element is a `vector<unsigned>` itself (i.e., not a basic type that can be communicated in MPI). Therefore, the function uses `MPI_Send/Recv` to perform the communications.

On the other hand, since `rank 0` sends the trajectories to the other ranks with a block partitioning schema (lines 59 to 73), the local vectors of probabilities can be accumulated using `MPI_Allgather`. Mind that this would not be possible if we decided to use a cyclic partitioning schema, which destroys the order of the trajectories with respect to the initial vector. In the second case, since we always want to return the global result from all ranks, a loop of `MPI_Bcast` would be needed.

Exercise 3 (8 points)

The following code models the informatic system through which a mountaineering association manages booking for an organized climbing trip. The association has rented a bus with capacity for 52 passengers, but participants to the trip will have to contribute to the bus renting by paying a small fare. Furthermore, the organization is making some climbing equipment available for those who should not own the required items, free of charge for the day.

The class `participation` models a group of friends that want to reserve a place for the trip. It contains information about the number of people in the group and how many of them need to borrow climbing gear as well. Everyone needs to wear a harness (imbrago) when climbing, but since belay devices (assicuratori per discesa) are used in pairs there is no need to borrow one device per person and it is enough to borrow one every two persons without gear.

The class `course` models groups of people that want to participate to the trip but have no experience. They will need to borrow all the gear and will be followed by two instructors per group. Instructors will travel on the bus as well and they will be the only ones in the group using belay devices.

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 participation bill for the first booking? (Value of `quota` printed in the first invocation of function `attempt_booking` at line 40)
2. How many belay devices (Value of `rented.second`) are needed to fulfill the needs of the participants of the second booking? (Second invocation of `attempt_booking` at line 41)
3. How many `belay_devices` are still available in `tripAvailable` after the third invocation of `attempt_booking`? (Line 42)
4. What is the value of `tripAvailable.bus_seats` before the execution of the main reaches the return? (Line 45)

Provided source code:

- **participation.h**

```
#ifndef EX3_P_H
#define EX3_P_H

#include <iostream>

class participation {
protected:
   unsigned participants;
   unsigned renting;

public:
   participation(unsigned p, unsigned r=0) : participants(p), renting(r) {
      if (r > p) {
         std::cout
               << "Error: there are more people renting equipment than participants"
               << std::endl;
         exit(1);
      }
   };
```

```
    participation(participation &p) : participation(p.participants) {};

    virtual unsigned getParticipants() const { return participants; };

    std::pair<unsigned, unsigned> getEquipmentToRent() const;

};

#endif //EX3_P_H
```

- **participation.cpp**

```
#include "participation.h"
#include <utility>

std::pair<unsigned, unsigned> participation::getEquipmentToRent() const {
    unsigned harnesses = renting;
    unsigned belay_devices = renting / 2;
    return std::make_pair(harnesses, belay_devices);
}
```

- **course.h**

```
#ifndef EX3_C_H
#define EX3_C_H

#include "participation.h"
#include <utility>

class course : public participation {

public:
    course(unsigned num) : participation(num, num){};

    unsigned getParticipants() const override;

    std::pair<unsigned, unsigned> getEquipmentToRent() const;
};

#endif //EX3_C_H
```

- **course.cpp**

```
#include "course.h"
#include <utility>

unsigned course::getParticipants() const { return participants + 2; }

std::pair<unsigned, unsigned> course::getEquipmentToRent() const {
    return std::make_pair(participants, 2);
}
```

- **main.cpp**

```
#include "course.h"
#include "participation.h"
#include <utility>

struct avail {
    unsigned bus_seats = 52;
7     unsigned harnesses = 18;
8     unsigned belay_devices = 12;
9  };
10
11 typedef struct avail availability;
12
13 void attempt_booking(const participation &p, availability &aval) {
14    unsigned seats = p.getParticipants();
15    std::pair<unsigned, unsigned> rented = p.getEquipmentToRent();
16
17    if (seats > aval.bus_seats || rented.first > aval.harnesses ||
18       rented.second > aval.belay_devices) {
19       std::cout << "Sorry, unsuccessful booking!" << std::endl;
20       return;
21    }
22
23    aval.bus_seats -= seats;
24    aval.harnesses -= rented.first;
25    aval.belay_devices -= rented.second;
26
27    unsigned quota = seats * 5;
28    std::cout << "The bill for your participation is " << quota << std::endl;
29 }
30
31 int main(int, char**) {
32    availability tripAvailable;
33
34    participation booking1(20);
35    course booking2(20);
36
37    course booking3(14);
38    participation booking4 = booking3;
39
40    attempt_booking(booking1, tripAvailable);
41    attempt_booking(booking2, tripAvailable);
42    attempt_booking(booking3, tripAvailable);
43    attempt_booking(booking4, tripAvailable);
44
45    return 0;
46 }
```

Solution 3

1. The first booking attempt is successful and involves 20 people, therefore the requested payment amount to $20 * 5 = 100$

2. Booking2 attempts to book a course for 20 people. Theoretically this would translate in the need to borrow only 2 belay devices regardless of the number of participants. However, since getEquipmentToRent is not a virtual method the superclass method is called and the result is 10

3. Given that booking2 requires more harnesses than those available and is rejected (returns at line 20), after confirming booking3 there are still 5 belay devices available in tripAvailable

4. Booking attempts for booking1, booking3 and booking4 are successful. Since getParticipants is a virtual method the result is $52 - (20(frombooking1) + 16(frombooking3) + 14(frombooking4)) = 2$