7th Recitation

Priority Queues

Instructions:

- Download the file `aed2324_p07.zip` from the course page and unzip it (it contains the `lib` folder, the `Tests` folder with the files `box.h`, `box.cpp`, `packagingMachine.h`, `packagingMachine.cpp`, `funPQProblem.h`, `funPQProblem.cpp` and `tests.cpp`, and the files `CmakeLists.txt` and `main.cpp`);
- In CLion, open a project by selecting the folder containing the files from the previous point.

1. For this Christmas season, Mr. Silva's store decided to innovate in order to make shipping its products more efficient. To do this, he bought an automatic packer that places objects in boxes according to the weight of each object and the remaining capacity of each box.

Suppose there is a set of boxes with the capacity (maximum weight they can hold) `boxCapacity`, identical to all the boxes and objects $o_1, o_2, ..., o_n$ with weights (`weight`) $w_1, w_2, ..., w_n$, respectively. The goal is to pack all the objects without exceeding the capacity of each box, but overall using as few boxes as possible. There is no predefined number of boxes. In fact, you will have to "create" new boxes as you go along.

Implement a program to solve this problem, using the following strategy:
- Start by placing the heaviest objects in a box first;
- Place an object in the box with the highest current load, but which still has available capacity to hold it.

To organize the objects by weight, store them in a priority queue (`priority_queue<Object>`), with the object with the greatest weight having the highest priority (i.e., at the top of the queue). In addition, it stores the various boxes also in a priority queue, named `boxes`, (`priority_queue<Box>`), with the box with the highest priority being the box with the least load still available (i.e., with the heaviest load).

```
class Object {
    unsigned id;
    unsigned weight;
public:
    Object(unsigned i, unsigned w);
    bool operator<(const Object& o1)
const;
    ...//
};

typedef std::stack<Object> StackObj;
class Box {
     StackObj objects;
     unsigned capacity;
     unsigned free;
public:
     Box(unsigned cap=10);
     bool operator<(const Box& b1) const;
     ...//
};
```

```
typedef std::priority_queue<Object> HeapObj;
typedef std::priority_queue<Box> HeapBox;

class PackagingMachine {
     HeapObj objects;
     HeapBox boxes;
     unsigned boxCapacity;
public:
     PackagingMachine(int boxCap = 10);

     ...//
};
```

* **Note:** you must carry out this exercise in the order of the subexercises.

1.1 Implement the member function described below:

```
unsigned PackagingMachine::loadObjects(std::vector<Object> &objs)
```

This function takes as an argument the palette of objects to be packed (vector `objs`). Only objects weighing less than or equal to the capacity of the boxes are loaded into the packer. The function returns the number of objects actually loaded into the packer, as follows:

- objects loaded in the `PackagingMachine` are removed from the `objs` vector;
- objects are loaded in the objects priority queue, according to their priority where the highest priority object is the heaviest.

1.2 Implement the member function described below:

```
Box PackagingMachine::searchBox(Object &obj)
```

This function searches the priority queue `boxes` for the next box with enough remaining load to hold the `obj` object given as its argument. If such a box exists, it removes it from the priority queue and returns it. If there is no box with enough free capacity for the `obj` presented as argument, it "creates" a new box and returns it.

* **Note:** This function does not explicitly place the `obj` in the box.

1.3 Implement the member function described below:

```
Box PackagingMachine::packObjects(Object &obj)
```

This function stores the objects in the `objects` queue in as few boxes as possible. It returns the number of boxes used. Consider that initially no boxes are being used, so obviously, one of the first operations will cause the creation of a new box.

1.4 Implement the member function described below:

```
std::stack<Object> PackagingMachine::boxWithMoreObjects() const
```

This function searches the priority queue of boxes, named `boxes`, for the box containing the largest number of objects and returns its contents (stack of objects). If there are no boxes in the `boxes` queue, the function returns an empty stack.

2. Consider the `FunPQProblem` class which has only static methods. Implement the member-function described below:

```
int FunPQProblem::minCost(const std::vector<int> &ropes)
```

Given a set of ropes of different lengths, it is necessary to connect them into a single rope. The cost of connecting two strings is equal to the sum of their lengths. The task is to connect the strings with minimum cost.

Suggestion: When choosing which strings to connect, choose the two strings with the shortest lengths. Use a priority queue. Think why this can not be done just by sorting the initial vector of ropes.

* **Note:** STL's `priority_queue` class implements a maximum priority queue. But the user-supplied `Compare` parameter can change the sorting direction, e.g., using `std::greater<T>` makes the smallest element appear in `top()`. For example, for an integer priority queue:

```
std::priority_queue<int, std::vector<int>, std::greater<int>>
```

Expected time complexity: O($n \times log\ n$)

Execution example:

input: *ropes* = `{4, 3, 2, 6}`
output: *result* = `29`
Connect ropes of length 2 and 3: `{4, 5, 6}`, *cost* `= 2 + 3 = 5`
Connect ropes of length 4 and 5: `{9, 6}`, *cost* `= 4 + 5 = 9`
Connect ropes of length 9 and 6: `{}`, *cost* `= 9 + 6 = 15`
Total cost = `5 + 9 + 15 = 29`