Exercice : Sac à dos

```cpp
/* ============================================================================
 *
 * Knapsack problem.
 *
 * Version: 1.0 (170901)
 * Author: J.-C. Chappelier (EPFL, 2017)
 *
 * ============================================================================
 */
#include <iostream>
#include <vector>
#include <algorithm> // pour sort()
using namespace std;

struct Object {
  double weight;
  double value;
};

/* ============================================================================
 * This is the first algorithm:
 * solve it naively, in the given order.
 */
void solve_naive(vector<Object> const& objects, double remaining_weight)
{
  double value(0.0);
  for (auto const& object : objects) {
    cout << "considering : w=" << object.weight << ", v=" << object.value;
    if (object.weight <= remaining_weight) {
      value += object.value;
      remaining_weight -= object.weight;
      cout << " --> taking";
    }
    cout << endl;
  }
  cout << "value = " << value << ", remaining weight = " << remaining_weight << endl;
}

// ============================================================================
bool higher_value(Object const& o1, Object const& o2)
{ return o1.value > o2.value; }

/* ============================================================================
 * This is the second algorithm: greedy.
 */
void solve_greedy(vector<Object> const& objects, double maximum_weight)
{
  vector<Object> sorted_objects(objects); // need a copy to sort
  sort(sorted_objects.begin(), sorted_objects.end(), higher_value);
  solve_naive(sorted_objects, maximum_weight);
}

/* ============================================================================
 * This is the third algorithm: try all.
 */
double solve_exact_recursive(vector<Object> const& objects, size_t start, double& remain
{
  double best_value(0.0);

  if ((not objects.empty()) and (start < objects.size())) {
    // 1) consider NOT taking first object
    double best_remaining_weight(remaining_weight);
    best_value = solve_exact_recursive(objects, start+1, best_remaining_weight);

    // 2) consider taking the first object, if possible
```

```
    if (objects[start].weight <= remaining_weight) {
      double weight(remaining_weight - objects[start].weight);
      const double value(objects[start].value + solve_exact_recursive(objects, start+1,


      // 3) keep the best of the two
      if (value > best_value) {
        best_remaining_weight = weight;
        best_value = value;
      }
    }
    remaining_weight = best_remaining_weight;
  }
  return best_value;
}

void solve_exact(vector<Object> const& objects, double remaining_weight)
{
  double value(solve_exact_recursive(objects, 0, remaining_weight));
  cout << "value = " << value << ", remaining weight = " << remaining_weight << endl;
}

// ============================================================================
int main()
{
  const vector<Object> objects({
      { 4.0,  8.0 },
      { 2.0,  3.0 },
      { 5.0,  7.0 },
      { 6.0, 10.0 },
  });
  constexpr double MAX(9.0);

  solve_naive (objects, MAX);
  solve_greedy(objects, MAX);
  solve_exact (objects, MAX);

  return 0;
}
```