TP5: Programmation fonctionnelle en C++

Objectifs pédagogiques :
- Utiliser des lambdas (tous les exercices)
- Mémoiser du code (Ex 1)
- Écrire des templates (Ex 1)
- Concevoir une classe d'itérateur (Ex 2 et 3)
- Écrire du code C++17 comme des affectations déstructurantes (Ex 1)
- Écrire du code C++20 avec des ranges (Ex 3, questions optionnelles)

L'exercice 4 (optionnel) reprend toutes les notions abordées dans un même problème.

★ Exercice 1: Mémoisation en C++.

Vous trouverez dans le fichier `memo-fibo.cpp` l'exemple de code étudié en cours pour la mémoisation générique de la suite de Fibonacci en C++11. Comme vous pouvez le constater, la table associative `memo` allouée à la ligne 7 n'est jamais libérée.

▷ **Question 1:** Modifiez le code fourni afin de corriger la fuite mémoire. Pour l'instant, `memoize` est une fonction templatée retournant une lambda. Le plus simple semble être que `memoize` soit une classe templatée évaluable. Le corps de la lambda doit être dans l'opérateur d'évaluation (`operator()`), et la table associative doit être allouée et détruite respectivement dans le constructeur et le destructeur de l'objet `memoize`. Si la table est un objet automatique, vous n'avez même pas besoin de gérer explicitement la mémoire.

▷ **Question 2:** Complétez le fichier `pascal.cpp` en proposant une solution permettant de mémoiser la fonction du triangle de Pascal. La difficulté est de réutiliser la classe `memoize`, en la modifiant pour qu'elle puisse être utilisée pour mémoiser des variables de deux paramètres. Il n'est pas demandé d'écrire une version de `memoize` fonctionnant à la fois pour Fibonacci et Pascal, mais bien de modifier une copie de cette classe.

▷ **Question 3:** Complétez le fichier `rodcutting.cpp` pour mémoiser le calcul de la solution optimale pour le problème de la découpe d'une barre. Dans ce problème, on un tableau de prix associant un prix à chaque longueur entre 1 et n. On veut découper une barre de longueur n en petites barres de façon à maximiser le prix de l'ensemble des barres obtenu. Le code fourni est fonctionnel, mais il recalcule toutes les solutions à chaque fois.

L'objectif de la question est d'adapter votre objet de mémoisation générique à ce nouveau problème, dont la spécificité est de retourner plusieurs paramètres sous la forme d'une `std::pair` et d'écritures C++17 nommées *affectation déstructurante* (structured binding en anglais).

Une fois la mémoisation en place, le nombre de calculs intermédiaires devrait grandement décroître.

★ Itérateurs, ranges et problème du sac à dos.

Nous allons écrire différentes solutions pour le problème du sac à dos. Nous allons modifier la solution bêtement récursive fournie pour mettre en place deux itérateurs. Le premier, `knap_eager`, calcule à l'avance toutes les solutions potentielles tandis que `knap_lazy` ne calculant que ce qui est nécessaire.

▷ **Découverte du code fourni**

Le fichier `knapsack.hpp` fourni contient trois classes. Le fichier est fonctionnel sans modification.
- `Item` : un objet que l'on peut prendre ou non, définie par deux valeurs : `value()` et `weight()`.
- `Problem` : Une instance du problème du sac à dos, définie par un vecteur d'items et une capacité maximale du sac à dos.
- `Solution` : Une solution, définie avec un vecteur de booléens dénotant si chaque item est pris ou non. Les solutions peuvent être partielles, quand leur vecteur de booléens est plus court que le nombre d'items dans le problème.

Le fichier `knap_recursive.cpp` contient une solution complète pour le problème du sac à dos. La fonction `recursive_solver` réalise un parcours en profondeur d'un arbre binaire. On commence la récursion avec une solution `current` vide passée en paramètre. À chaque étage, on fait un appel récursif une nouvelle solution où l'objet est pris, puis avec une solution où l'objet n'est pas pris (respectivement avec `take_one()` et `leave_one()`). On s'arrête et on remonte dans l'arbre quand on arrive à une solution complète (i.e. non-partielle) et on sauvegarde la meilleure solution croisée en chemin. Vous pouvez lancer ce code pour observer son résultat.

```
$ ./knap_recursive
[tests avec l'instance 0, celle par défaut]
$ ./knap_recursive 1
[tests avec l'instance 1]
```

★ **Exercice 2: Itérateur de base.**

L'objectif de cet exercice est d'écrire votre premier itérateur. On souhaite pouvoir l'utiliser ainsi :

```
std::vector<Solution> solutions;
for (auto s: EagerSearch(p))
  if (s.is_valid())
    solutions.push_back(s);
std::sort(solutions.begin(), solutions.end(),
          [](Solution a, Solution b){return a.value()>b.value();});
auto best = solutions.front();
```

Les premières lignes de ce code sont équivalentes à la forme étendue suivante :

```
auto searcher =  EagerSearch(p);
for (EagerSearch::iterator it = searcher.begin(); it != searcher.end(); it++) {
  auto s = *it;
  solutions.push_back(s);
}
std::sort(...
```

Un itérateur est donc une classe stockant l'état actuel du parcours. La collection doit fournir deux méthodes : `EagerSearch::begin()` et `EagerSearch::end()`, qui renvoient respectivement un itérateur sur le début de la collection et un itérateur sur la fin de la collection. La classe `iterator` doit fournir trois opérateurs :

- `operator++` qui fait passer l'itérateur à l'élément suivant de la collection.
- `operator*` qui renvoie l'élément sur lequel l'itérateur pointe actuellement.
- `operator!=` qui est principalement utilisé pour mesurer si l'itérateur courant (initialement égal à `begin()` et sur lequel on fait ++) est devenu égal à l'itérateur indiquant la fin de collection.

Si nous définissions un itérateur sur un vecteur, il suffirait de stocker en interne une référence au vecteur parcouru, et un numéro d'index indiquant où nous en sommes. `operator++` incrémenterait l'index, `operator*` retournerait l'élément du vecteur à cet index, `operator!=` comparerait les numéros d'index. `begin()` créerait un itérateur d'index 0 tandis que `end()` créerait un itérateur dont l'index pointe juste après la taille du tableau.

Mais notre itérateur ne parcourt pas un tableau : il représente un parcours d'arbre en profondeur. Un simple indice numérique n'est certainement pas suffisant pour stocker l'état actuel du parcours pour le reprendre plus tard (cf. exercice suivant). Pour simplifier dans cet exercice, nous allons précalculer récursivement tous les éléments à parcourir dans le constructeur de l'itérateur, et les autres opérateurs utiliseront ces solutions stockées dans un vecteur interne à la classe `EagerSearch::iterator`.

▷ **Question 1:** Écrivez cet itérateur naïf en complétant la classe `EagerSearch` fournie dans `knap_eager.cpp` Le constructeur utilise la fonction `generate_all()` pour calculer toutes les solutions et les placer dans le vecteur `solutions_` (`generate_all()` est écrit comme une lambda car c'est la façon de faire des fonctions locales en C++). Ensuite, l'`operator*` de l'itérateur retourne le dernier élément du vecteur tandis que l'`operator++` passe à l'élément suivant en retirant un élément du vecteur.

Il faut également définir les méthodes `EagerSearch::begin()` et `EagerSearch::end()`. Le plus difficile est de déterminer une convention pour l'itérateur de fin de parcours, `EagerSearch::end()`. On pourrait tricher sur l'`operator!=` pour détecter si l'utilisateur teste la fin de parcours, mais c'est une mauvaise idée car l'utilisateur pourrait utiliser notre code autrement qu'indiqué ci-dessus. Il est préférable d'avoir de vrais opérateurs d'égalité et d'inégalité, et il faut trouver une autre idée.

On constate que nos itérateurs ont fini quand leur vecteur de solutions est vide. On va donc faire en sorte que `end()` commence avec un vecteur vide. C'est à ça que sert le paramètre optionnel `empty_iterator` du code fourni. S'il est vrai, aucune solution ne doit être ajoutée au vecteur `solutions_`.

▷ **Question 2:** Optimisez votre itérateur en ne parcourant pas les solutions invalides.

▷ **Question 3:** Testez votre code, et vérifiez que vous trouvez les mêmes solutions avec le code récursif.

★ **Exercice 3: Itérateur paresseux.**

La solution précédente est très mauvaise, car elle consomme énormément de mémoire. On souhaite maintenant réaliser un itérateur qui sauvegarde l'état actuel du parcours récursif afin de pouvoir le geler dès qu'on trouve une solution, et reprendre ce parcours quand l'itérateur doit passer à l'élément suivant.

Pour cela, nous allons dérécursiver la fonction de génération en manipulant explicitement une pile d'appels. L'état interne de notre itérateur est donc donné maintenant par une pile des solutions partielles en cours de fabrication. Initialement, la pile contient la solution vide. Il faut faire une opération supplémentaire dans le constructeur de l'itérateur, car l'invariant de la pile est que si elle contient au moins un élément, alors l'élément au sommet est une solution complète.

Si l'élément au sommet n'est pas une solution complète, on peut avancer le calcul de la manière suivante : Soit `e` l'élément au sommet. On dépile `e`. On empile `e.leave_one()` et on empile `e.take_one()`. Il peut être nécessaire de réaliser cette opération plusieurs fois, jusqu'à atteindre l'invariant sus-cité.

Ainsi dans l'exemple ci-dessous pour 4 éléments, le constructeur commence par ajouter `(....)` à la pile (c'est à dire la solution vide). Mais comme cela ne respecte pas l'invariant, le constructeur applique l'opération décrite au paragraphe précédent : il remplace tout d'abord `(....)` par `(N...)` et `(Y...)` (ligne 1 à 2). Comme l'invariant n'est toujours pas respecté, il faut encore remplacer `(Y...)` par `(YN..)` et `(YY..)` (ligne 2 à 3). Après plusieurs étapes, on arrive à l'état de la ligne 5, où l'invariant est respecté. Le constructeur a terminé son initialisation ; l'opérateur `*` trouvera bien une solution valide au sommet de la pile.

Plus tard, chaque fois que l'opérateur ++ élimine le sommet de la pile, il doit refaire les mêmes opérations jusqu'à ce que l'invariant soit respecté à nouveau. Il est donc avantageux de définir une méthode privée permettant de factoriser le code entre le constructeur et l'opérateur ++.

Valeurs successives de la pile pour 4 éléments (on dépile/empile à droite)

```
1  (....)
2  (N...)(Y...)
3  (N...)(YN..)(YY..)
4  (N...)(YN..)(YYN.)(YYY.)
5  (N...)(YN..)(YYN.)(YYYN)(YYYY) # État à la fin du constructeur : invariant respecté.
6  (N...)(YN..)(YYN.)(YYYN)       # operator++() a éliminé (YYYY) ; Invariant respecté.
7  (N...)(YN..)(YYN.)             # operator++() a éliminé (YYYN) ; Invariant violé!
8  (N...)(YN..)(YYNN)(YYNY)       # Invariant respecté.
9  (N...)(YN..)(YYNN)             # operator++() a éliminé (YYNY) ; Invariant respecté.
10 (N...)(YN..)                   # operator++() a éliminé (YYNN) ; Invariant violé!
11 (N...)(YNN.)(YNY.)
12 (N...)(YNN.)(YNYN)(YNYY)       # Invariant respecté.
13 (N...)(YNN.)(YNYN)             # operator++() a éliminé (YNYY) ; Invariant respecté.
14 (N...)(YNN.)                   # operator++() a éliminé (YNYN) ; Invariant violé!
15 (N...)(YNNN)(YNNY)             # Invariant respecté.
16 (N...)(YNNN)                   # operator++() a éliminé (YNNY) ; Invariant respecté.
17 (N...)                         # operator++() a éliminé (YNNN) ; Invariant violé!
18 (NN..)(NY..)
19 (NN..)(NYN.)(NYY.)
20 (NN..)(NYN.)(NYYN)(NYYY)       # Invariant respecté.
21 (NN..)(NYN.)(NYYN)             # operator++() a éliminé (NYYY) ; Invariant respecté.
22 (NN..)(NYN.)                   # operator++() a éliminé (NYYN) ; Invariant violé!
23 (NN..)(NYNN)(NYNY)             # Invariant respecté.
24 (NN..)(NYNN)                   # operator++() a éliminé (NYNY) ; Invariant respecté.
25 (NN..)                         # operator++() a éliminé (NYNN) ; Invariant violé!
26 (NNN.)(NNY.)
27 (NNN.)(NNYN)(NNYY)             # Invariant respecté.
28 (NNN.)(NNYN)                   # operator++() a éliminé (NNYY) ; Invariant respecté.
29 (NNN.)                         # operator++() a éliminé (NNYN) ; Invariant violé!
30 (NNNN)(NNNY)                   # Invariant respecté.
31 (NNNN)                         # operator++() a éliminé (NNNY) ; Invariant respecté.
32                                # operator++() a éliminé (NNNN) ; État final.
```

À la fin, les différentes occurrences de l'opérateur ++ ont bien parcouru les 16 solutions attendues.

▷ **Question 1:** Complétez `knap_lazy.cpp` pour créer un tel itérateur paresseux capable de sauvegarder l'état d'une recherche récursive puis de la reprendre au besoin. Ne vous préoccupez pas de performance pour l'instant, mais vérifiez que votre solution trouve bien les solutions attendues et que `valgrind` ne trouve pas de problème avec votre implémentation. La clé du succès passe par la simplicité de votre code : pas de complexité inutile.

▷ **Question 2:** Optimisez votre itérateur paresseux en ne parcourant pas les solutions invalides.

▷ **Question 3:** Il est très agaçant de constater que le code de test fait parcourir toutes les solutions de votre itérateur paresseux avant de les trier. Cela réduit à néant les gains en mémoire par rapport à l'itérateur de la section précédente !

Notre itérateur doit certes retourner les solutions valides au fur et à mesure de leur découverte, ce qui l'empêche de les trier pour trouver la meilleure solution. En revanche, il est possible d'optimiser énormément la recherche en modifiant notre itérateur afin qu'il ne retourne jamais de solution moins bonne que la meilleure rencontrée jusque là. Avec cette optimisation, ma solution ne retourne que 8 solutions parmi les 1040334 solutions valides de l'instance `#1`. Cela réduit le temps de calcul d'un ordre de grandeur.

▷ **Question 4: (optionnelle)** Mise en place d'un `ranges::zip`.
Réécrivez les méthodes `Solution::weight()` et `Solution::is_valid()` définies dans le fichier `knapsack.hpp` sous forme de `ranges::zip`. Il sera nécessaire pour celà d'installer la bibliothèque ranges-v3 puisque le `zip` ne sera pas disponible dans le standard avant le C++23 : `sudo apt install librange-v3-dev`

Il s'agit de réécrire la boucle de ces fonctions sous forme d'un `zip`, tubé dans un filter (pour ne prendre que les objets sélectionnés), tubé dans un `transform` pour ne garder que la valeur de chaque objet. La vue correspondante sera passée à un `accumulate` pour faire la sommation.

▷ **Question 5: (optionnelle) les ranges C++20.** Augmenter notre itérateur pour permettre son usage dans un `range` C++20 est attirant, mais cela s'avère extrêmement compliqué en pratique. Les *concepts* C++ sont

inévitables, bien que cette notion soit hors programme de notre cours. Voici quelques liens si vous souhaitez vous attaquer à ce problème malgré tout. Je pense personnellement que C++ atteint ici ses limites.

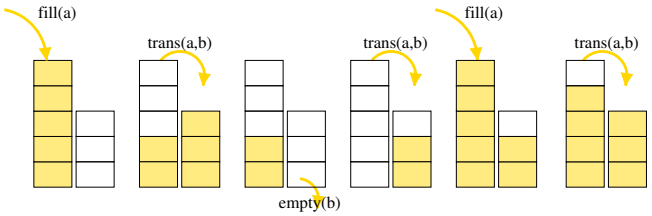
— Implémentation d'un itérateur basé sur un vecteur, en C++17.
— Définition d'un conteneur de données pouvant être parcouru avec un range, en C++20.
— Itérateur sur un parcours d'arbre, en C++17.

★ **Exercice 4: Problème d'application : les transvasements (optionel).**

Dans le problème des transvasements (water pouring puzzle ou Decanting problem en anglais), on dispose d'un certain nombre de récipients dont on connaît la capacité, et d'une fontaine d'eau. On cherche quels sont les transvasements à réaliser pour passer faire en sorte que l'un des récipients contienne une quantité d'eau donnée. Seules les opérations suivantes sont autorisées : (A) Remplir complètement un récipient depuis la fontaine ; (B) Vider complètement un récipient dans la fontaine ; (C) Transvaser un récipient dans un autre jusqu'à ce que la source soit complètement vide ou que la destination soit complètement pleine.

▷ **Exemple.** On suppose avoir deux récipients de capacité respective 5 et 3. On veut mesurer un volume de 4. Dans la situation initiale, notée $(0, 0)$, les deux récipients sont vides. Voici une solution en 6 opérations.

— Remplir A à la fontaine : (5,0)
— Transvaser A dans B : (2,3)
— Vider le contenu de B : (2,0)
— Transvaser A dans B : (0,2)
— Remplir A à la fontaine : (5,2)
— Transvaser A dans B : (4,3)
— On a bien 4 unités dans A.



Comme dans le cas du sac à dos, il existe des méthodes spécifiques pour résoudre ce problème (ici, en utilisant la théorie des graphes), mais on peut aussi utiliser un bête backtracking comme précédement. L'objectif de cet exercice est de réutiliser toutes les idées vues aux exercices précédents.

▷ **Code fourni.** Le fichier `waterpouring.hpp` défini une classe `Problem` et une classe `Solution` comparables à ce que nous avions pour le problème du sac à dos. Générer de nouvelles solutions se fait en tentant tous les transvasements de l'un des récipients dans un autre.

Le fichier `waterpouring.cpp` donne une fonction de recherche exhaustive de solution, et une fonction principale testant cette fonction sur l'instance choisie. Comme ce problème peut boucler (par exemple en transvasant le premier récipiant dans le second, avant de recommencer la même opération à l'infini), le code fourni n'explore le problème que jusqu'à une profondeur donnée. Par exemple `./waterpouring 1` teste le code fourni sur la deuxième instance prédéfinie.

Le code fourni corrige la première instance connue en quelques secondes, mais il échoue après plusieurs minutes sur la seconde instance, car il n'existe aucune solution de longueur inférieure à la borne de profondeur. On pourrait augmenter la valeur de la borne pour trouver les solutions existantes, mais il est bien plus efficace d'optimiser la recherche pour espérer résoudre les instances fournies.

▷ **Question 1:** La première idée est de réutiliser une construction `memoize` comme à l'exercice 3, mais au lieu de stocker les solutions partielles déjà calculées, il faut s'interdire d'explorer des branches déjà explorées. Il faut donc modifier `memoize` pour que `memoize::memo` soit de type `std::unordered_set`. Avant toute invocation à la fonction lambda, on regardera si la clé proposée est déjà dans l'ensemble. Si non, on l'y ajoute avant d'invoquer la lambda. Si oui, il faut renvoyer un élément neutre au lieu d'invoquer la lambda. Le constructeur de l'objet `memoize` doit donc prendre un élément neutre en second paramètre. Dans cet exercice, on pourra utiliser `std::vector<Solution>{}` à cet effet (un vecteur de taille nulle typé correctement).

▷ **Question 2:** Trouvez d'autres optimisations pour trouver le plus rapidement possible une solution valide. Pour information, mon code résout chacune des instances fournies en moins de cinq secondes.

Si je cherche une solution la plus courte possible (c'est-à-dire, celle dont la réalisation demande le moins d'étapes), certaines instances occupent mon code pendant plusieurs minutes. La dernière instance proposée est vraiment difficile à résoudre avec un parcours en profondeur. Ma solution en largeur prend plus de temps à compiler qu'à résoudre la plupart des instances.

▷ **Question 3:** Quelle est la pire instance de 3 récipients que vous pussiez trouver ? C'est à dire l'instance dont la plus courte solution est la plus longue possible.