TD 1 : recherche d'éléments, implantation

Initiation au développement

BUT informatique, première année

Ce TD vise à mettre en œuvre les concepts vus dans le cours *ALG2 : recherche d'éléments*. Pour cela nous allons créer un module et l'utiliser pour développer une bibliothèque de fonctions Go permettant de chercher des éléments dans des tableaux. Nous mettrons en place des jeux de tests pour chacune de nos fonctions et nous évaluerons leurs performances.

Un texte avec cet encadrement contient une information importante.

Un texte avec cet encadrement est une remarque.

Un texte avec cet encadrement est un travail que vous avez à faire.

1 Organisation de la bibliothèque

Nous allons coder nos fonctions au sein d'une bibliothèque qui aura vocation à s'étendre au fil du semestre : elle contiendra à la fin l'implantation de tous les algorithmes vus en CM. Cette bibliothèque prendra la forme d'un module contenant plusieurs paquets (un pour chaque algorithme ou groupe d'algorithmes similaires).

À partir de maintenant, lorsque vous faites des exercices en TP vous avez le droit d'utiliser les fonctions de votre bibliothèque pour vous aider. Vous pouvez aussi enrichir votre bibliothèque avec des fonctions utilitaires développées pour un exercice, afin de les réutiliser pour en faire d'autres.

Le nom de votre module et ceux de vos paquets doivent donc avoir du sens, indiquer ce qu'on trouve dans votre bibliothèque et les différentes fonctionnalités qu'elle fournit.

Choisissez un nom pour votre module puis initialisez-le.

Dans ce premier TD nous allons implanter quatre fonctionnalités :
— la recherche d'éléments dans un tableau d'entiers,
— la recherche de minimum dans un tableau d'entier,
— la sélection d'éléments dans un tableau d'entiers,
— la recherche d'éléments par dichotomie dans un tableau trié d'entiers.

Plusieurs organisations de ces fonctionnalités en paquets sont imaginables, il va vous falloir en choisir une.

Proposez et justifiez une organisations en paquets des quatre fonctionnalités ci-dessus. Créez les dossiers pour recevoir ces paquets.

2 Implantation et test d'une fonctionnalité

Ce travail est à faire pour chacune des quatre fonctionnalités à implanter : recherche d'éléments, recherche de minimum, sélection d'éléments, recherche d'éléments par dichotomie.

Le développement d'une fonctionnalité se fait par des aller-retours réguliers entre la définition de tests et le codage de la fonctionnalité : les tests peuvent nous indiquer des erreurs dans le codage, la réflexion sur le codage peut nous suggérer de nouveaux tests à réaliser.

2.1 Phase de réflexion

Avant de se lancer dans le codage, il est important d'avoir une phase de réflexion : bien définir les entrées et sorties de notre (ou de nos) fonction(s) et réfléchir aux premiers tests nécessaires.

Parfois, parmi les sorties d'une fonction, on peut avoir besoin de retourner une erreur (par exemple si les entrées ne sont pas exploitables : tableau qui vaut nil, indice négatif dans un tableau, etc). Pour cela, on peut utiliser la fonction New du paquet errors [1] (avec un s) et définir une variable de type error (sans s).

Définissez les paramètres et les valeurs de retour de la fonction qui implantera la fonctionnalité que vous développez. En particulier, demandez-vous s'il faut pouvoir retourner une erreur ou si on pourra toujours retourner une valeur correcte quelles que soient les paramètres de la fonction ?

Définissez un premier jeu de tests pour votre fonction. Demandez-vous pour cela quelles sont les valeurs limites à tester (tableau vide, élément absent ou présent, etc).

2.2 Codage et test

Une fois la phase de réflexion préliminaire réalisée, on peut commencer à coder nos fonctions. Puis les tester avec notre jeu de tests.

Si nos fonctions sont un peu complexes (plus de 5 à 10 lignes de codes) il est important de faire des petits tests au fur et à mesure de leur développement. On ne parle pas ici de définir un jeu de tests, mais d'essayer les fonctions sur des valeurs simples et de faire des affichages à des passages clés du code pour bien s'assurer que ce qui se passe est ce qu'on avait prévu. Pour cela vous pouvez éventuellement créer un main à la racine de votre module et l'utiliser pour appeler vos fonctions.

Codez votre fonctionnalité puis, une fois que vous pensez avoir terminé, testez-là avec votre jeu de tests. Si vous vous rendez compte en codant que vous avez oublié des tests importants, ajoutez-les à votre jeu de tests immédiatement.

2.3 Couverture

Comme nous l'avons vu en cours, ce n'est pas parce que notre fonctionnalité passe tous nos tests qu'elle est correcte. On peut oublier des tests importants. Les fonctionnalités de la commande go test

1. `https://pkg.go.dev/errors`

permettent de détecter automatiquement une partie de ces oublis en évaluant la couverture du code, c'est-à-dire en indiquant les parties du codes (fonctions, branches de conditionnelles) qui ne sont utilisées par aucun test. Un code non couvert par les tests peut être inutile à la fonctionnalité (et dans ce cas il faut le supprimer) ou bien il doit être testé (et dans ce cas il faut ajouter des tests dans le jeu de tests).

Calculer la couverture. Pour calculer la couverture d'un jeu de tests il suffit d'utiliser la commande go test avec l'option -cover, c'est-à-dire de faire go test -cover. On voit alors un pourcentage s'afficher à l'écran, qui indique la proportion de notre code qui est couvert par les tests.

> Calculez la couverture de votre code par vos tests.

Visualiser la couverture. Si le code n'est pas complètement couvert par les tests, il peut être intéressant de savoir quelle partie de celui-ci n'est jamais utilisée. Ceci peut se faire en utilisant la commande go test -coverprofile=couverture.out, qui crée un fichier couverture.out contenant des informations sur le code non couvert.

On peut directement lire ce fichier, mais ce n'est pas toujours pratique, surtout pour des codes un peu longs. Une autre solution consiste à utiliser ensuite la commande go tool cover -html=couverture.out qui va permettre de visualiser dans un navigateur internet la couverture du code.

> Si nécessaire, visualisez la couverture de votre code par vos tests. Déterminez alors si le code non couvert est utile ou non et ajoutez si besoin des cas de test pour le couvrir.

3 Évaluation de performances

Nous avons vu en cours que la recherche dans un tableau trié est plus efficace en général par dichotomie (1) que si on la fait linéairement (2). On peut, aussi, dans le cas d'un tableau trié, améliorer un peu la recherche linéaire en s'arrêtant dès qu'on trouve une valeurs supérieure à la valeur cherchée (3). L'objectif de cette dernière partie du TD est de confirmer expérimentalement que (1) et (3) sont plus efficaces que (2) et de trouver si (1) est plus ou moins efficace que (3).

> Vous avez normalement codé la méthode (1) et la méthode (2). Codez maintenant la méthode (3) (il s'agit de faire une petite adaptation de la méthode (2)) en choisissant judicieusement le paquet dans lequel la placer.

> Concevez et ajoutez des benchmarks à vos jeux de tests pour pouvoir comparer les trois méthodes. Testez puis concluez.