Bienvenue dans l'examen blanc de mi-semestre !

Règles du jeu

Le but de cet examen blanc est de faire un point à mi-parcours et de vous permettre de vous positionner par rapport à l'avancement et aux attendus du cours : **il n'est pas noté**.

Pour s'entrainer on va faire semblant d'être en mode examen :

- le réseau est coupé, sauf pour le site du cours https://algo-prog-1.pages.ensimag.fr/web/ ainsi que le site de la documentation Go https://pkg.go.dev/ ;
- vous travaillez en local sur la machine : vous n'avez pas accès à votre compte informatique et donc à vos données ;
- aucun document ni support électronique ne sont autorisés, sauf la feuille A4 ci-dessous ;
- vous avez droit à une feuille A4 **manuscrite** recto verso de notes libres ;

Sur le bureau de la machine, vous verrez deux icônes intéressantes :

- celle qui s'appelle « envoyer / send » sert à sauvegarder votre code au fur et à mesure de l'examen, au cas où votre machine plante (**ce qui est déjà arrivé**) : il est recommandé de cliquer dessus régulièrement et au moins à chaque fois que vous finissez une fonction ; évidemment, **il faut avoir sauvegardé dans votre éditeur de texte avant de s'en servir**, ce qu'on vous recommande de faire régulièrement aussi.
- celle qui s'appelle « envoyer et finir / send and exit » ne doit être utilisée qu'à la fin de la session : elle va sauvegarder une dernière fois votre code et arrêter proprement la machine ; avant de cliquer dessus, fermez proprement votre éditeur de texte pour être sûr que tout est bien enregistré.

Mais **AVANT DE TERMINER LA SESSION**, vous devez recopier votre rendu sur une autre machine, car on va retravailler dessus pendant la séance suivante. Pour cela, tapez simplement la commande suivante dans un terminal, en remplaçant `login` par votre login habituel :

```
scp -r ${HOME}/exam login@ensipcetu.ensimag.fr:examen_blanc
```

Crible d'Ératosthène

On va implanter dans cet exercice le fameux crible d'Ératosthène (célèbre informaticien grec du troisième siècle avant notre ère). Le crible d'Ératosthène est un algorithme permettant de lister tous les nombres premiers inférieurs ou égaux à une valeur maximale donnée.

Le principe de cet algorithme est très simple :

- on demande à l'utilisateur la valeur maximale `valMax` ;
- on alloue un *slice* (qu'on appellera `lesPremiers`) de `valMax` booléens, et on initialise toutes ses cases à `true` : ce *slice* sera en pratique utilisé comme un tableau statique (comme on a fait pendant la séance sur les tris par exemple) et vous ne devez jamais modifier sa taille une fois qu'il est créé ;
- on affecte la première case de ce tableau à `false`, car 1 n'est pas un nombre premier ;
- on pose que `prem` vaut 2 (le premier nombre premier) et on parcourt `lesPremiers` pour mettre à `false` toutes les cases dont le numéro est multiple de `prem` : 4, 6, 8, etc. ne sont pas des nombres premiers, on les « barre » dans le tableau ;
- on passe `prem` au plus petit entier supérieur qui n'a pas été barré et on répète le processus :
  - le plus petit entier supérieur à 2 et non barré est `prem = 3` : on barre tous les multiples de 3 dans le tableau ;
  - le plus petit entier supérieur à 3 et non barré est `prem = 5` (4 est un multiple de 2, il a déjà été barré) : on barre tous les multiples de 5 dans le tableau ;
  - et ainsi de suite.
- on peut s'arrêter dès que `prem` atteint $\sqrt{valMax}$, car on est alors sûr d'avoir déjà barré tous les nombres composés plus grands.

On détaille ci-dessous les différentes étapes à réaliser pour implanter cet algorithme : vous devez travailler dans le sous-répertoire `exo1` en complétant le fichier `crible.go` fourni.

On rappelle qu'en Go les *slices* sont forcément indicés à partir de 0 : la case numéro 1 (la première) d'un tableau à l'indice 0, la case numéro 2 (la deuxième) a l'indice 1, etc.

Initialisation du tableau

Commencez par écrire une fonction `initialiser(taille int) []bool` dont le but est d'allouer un *slice* de booléens de la `taille` voulue, d'initialiser ses cases à `true` (sauf la première qui devra être initialisée à `false`) et de renvoyer ce tableau à la fonction principale. Vous aurez vraisemblablement besoin de la fonction `make` (documentation) pour répondre à cette question.

Filtrage des multiples d'un nombre donné

Implantez maintenant une fonction `filtrer(lesPremiers []bool, prem int)` dont le rôle est d'affecter à `false` toutes les cases du tableau `lesPremiers` dont le numéro est un multiple de `prem`.

Affichage du résultat

Écrivez ensuite une fonction `afficher(lesPremiers []bool, valMax int)` qui affiche les numéros des cases contenant `true`, sous le format suivant (en supposant que `valMax == 10`) : `Les nombres premiers inférieurs ou égaux à 10 sont : 2 3 5 7`.

Programme principal

Implantez enfin la fonction `main` qui sera le point d'entrée du programme et devra donc demander à l'utilisateur la valeur maximale, puis utiliser les fonctions précédentes pour implanter l'algorithme du crible d'Ératosthène.

Votre programme devra bien vérifier que la valeur entrée par l'utilisateur est un entier supérieur ou égal à 2. Dans le cas contraire, on arrêtera immédiatement le programme avec un message d'erreur adapté sur *stderr*, par exemple en appelant `log.Fatal` (documentation) sur une erreur pouvant être créée par `errors.New` (documentation).

Vous aurez vraisemblablement besoin de créer un `Scanner` pour lire sur `os.Stdin`, en utilisant la fonction `bufio.NewScanner` (documentation) et les méthodes `Scan` (documentation) et `Text` (documentation) à appeler sur le `Scanner`.

On rappelle que la fonction `strconv.Atoi` (documentation) permet de convertir une chaîne de caractères en entier, si c'est possible (et on rappelle aussi qu'elle est susceptible de renvoyer une erreur qui devra être gérée proprement).

Enfin, des fonctions mathématiques intéressantes comme `math.Floor` (documentation) et `math.Sqrt` (documentation) pourront vous être utiles.

Tri de la crêpe

Le tri de crêpe est un algorithme de tri de tableaux inventé par un informaticien affamé et muni d'une spatule.

Le principe général de ce tri est similaire à celui du tri par sélection du maximum. À chaque pas de l'itération, on considère que le tableau est découpé en deux parties :

- la partie déjà triée à droite du tableau, entre les indices $[sup..taille[$, où *taille* est le nombre d'éléments du tableau complet ;
- la partie à trier à gauche du tableau, entre les indices $[0..sup[$.

À chaque pas de l'itération, on va chercher la valeur maximale du sous-tableau non trié pour la mettre à sa place, c'est-à-dire à l'indice $sup - 1$, puis déplacer *sup* d'une case vers la gauche.

Tant que le sous-tableau non trié contient au moins deux éléments, on va donc effectuer les actions ci-dessous.

On commence par rechercher l'indice de la valeur maximale dans le sous-tableau non trié.

Si cet indice est égal à $sup - 1$, la valeur maximale est en fait déjà à sa place et on passe au pas suivant de l'itération.

Sinon, et uniquement si cet indice est différent de zéro, on *retourne* le sous-sous-tableau correspondant à la tranche du tableau entre les indices zéro et l'indice de la valeur maximale inclus.

Retourner une tranche de tableau consiste simplement à inverser tous ses éléments : par exemple, si on retourne le tableau `[1 2 3 4]`, on obtient le tableau `[4 3 2 1]`.

Après ce premier retournement, la valeur maximale est donc placée dans la case d'indice zéro.

On *retourne* ensuite tout le sous-tableau non trié : la valeur maximale est donc maintenant dans la case d'indice $sup - 1$ et on peut passer au pas suivant de l'itération.

Travail demandé

Vous devez écrire un programme qui va lire un flot d'entiers signés sur *stdin* et afficher le tableau dynamique (*slice*) contenant les mêmes entiers triés par ordre croissant sur *stdout*.

On fournit le squelette de base du programme dans le fichier `tri_crepe.go` dans le sous-répertoire `exo2`.

On vous recommande de découper votre code en fonctions, pour pouvoir les tester plus facilement et rendre le code plus clair. Par exemple, il est plus clair d'écrire une fonction qui retourne une tranche de tableau, une autre qui recherche l'indice de la valeur maximale dans une tranche de tableau, et de les appeler dans votre fonction de tri.

La fonction qui va lire les entiers depuis *stdin* et renvoyer le tableau devra bien gérer les contraintes suivantes et gérer les erreurs correctement :

- les entiers dans le flot d'entrée sont séparés par un ou plusieurs espaces ;
- les lignes sont séparées par un ou plusieurs retours à la ligne (autrement dit, il peut y avoir des lignes vides ou ne contenant que des espaces) ;
- les caractères différents d'un chiffre, d'un signe moins, d'un espace ou d'un retour chariot provoqueront une erreur que vous devrez gérer correctement.

La fonction `strings.Fields` (documentation) permet de fractionner une chaîne de caractères en un tableau de *strings* correspondant aux sous-chaînes séparés par un ou plusieurs « caractères d'espacement » (c'est-à-dire des espaces, tabulations, retours à la ligne...).

La fonction `append` (documentation) permet d'ajouter des éléments à la fin d'un *slice* créé avec la fonction `make` (documentation). Elle s'utilise très simplement en écrivant par exemple `tab = append(tab, val)` pour ajouter la valeur `val` à la fin du *slice* `tab` qui va grandir automatiquement pour permettre cet ajout : vous pouvez donc créer le *slice* initialement avec une taille de 0 (`tab := make([]int, 0)`). Si cette idée de tableau pouvant changer de taille vous parait mystérieuse, vous pouvez aussi allouer un *slice* de taille 1000 dès le début de la fonction et gérer un indice `premiereCaseVide` que vous incrémenterez à chaque ajout d'une valeur dans le tableau.

Vous pourrez mettre au point votre programme en tapant directement au clavier les entiers, ou en utilisant les petits exemples ci-dessous.

- fichier avec zéro entier (ex_zero.txt) et le résultat attendu (res_zero.txt) ;
- fichier avec un entier (ex_un.txt) et le résultat attendu (res_un.txt) ;
- fichier avec deux entiers (ex_deux.txt) et le résultat attendu (res_deux.txt) ;
- fichier avec trois entiers (ex_trois.txt) et le résultat attendu (res_trois.txt) ;
- fichier avec quatre entiers (ex_quatre.txt) et le résultat attendu (res_quatre.txt) ;
- fichier avec cinq entiers (ex_cinq.txt) et le résultat attendu (res_cinq.txt) ;
- fichier avec mille entiers (ex_mille.txt) et le résultat attendu (res_mille.txt).

On rappelle que vous pouvez utiliser des redirections Unix pour lire les entiers directement dans un fichier et écrire le résultat dans un autre fichier, par exemple : `go run tri_crepe.go < ex_cinq.txt > sortie.txt`, puis comparer votre résultat avec celui attendu avec la commande `diff sortie.txt res_cinq.txt`.

Enfin, s'il vous reste du temps, vous implanterez une fonction de test automatique qui effectue beaucoup (100 par exemple) de tris sur des tableaux de « grandes » tailles (entre 0 et 500), remplis avec des entiers tirés aléatoirement (entre -10 et 10 par exemple, en utilisant la fonction `rand.Intn` (documentation), et compare le résultat obtenu avec la fonction `slices.Sort` (documentation) fournie par Go. Les fonctions `copy` (documentation) et `slices.Equal` (documentation) vous seront vraisemblablement utiles. Vous pouvez envoyer les sorties de cette fonction de test sur *stderr* si vous ne voulez pas perturber les affichages attendus.

Avant de partir...

Vous allez re-travailler ce sujet pendant la prochaine séance de TP. Pour éviter d'avoir à tout refaire, tapez la commande suivante dans le terminal, en remplaçant `login` par votre login habituel :

```
scp -r ${HOME}/exam login@ensipcetu.ensimag.fr:examen_blanc
```