Travail pratique: Reed-Solomon

1 Préambule

Lisez tout l'énoncé avant de commencer à programmer!!!

Ce travail pratique est organisé en **trois parties** traitant de la notion de librairie dont le but final est l'implémentation du code de Reed-Solomon. Les parties sont les suivantes :

1. Implémentation d'une librairie de fractions et d'un programme l'utilisant.
2. Implémentation d'une librairie de polynômes à coefficients fractionnaires et d'un programme l'utilisant.
3. Utilisation des librairies de fractions et de polynômes pour une implémentation d'un code de Reed-Solomon.

1.1 Généralités

Dans ce TP, il y a une grande quantité de fonctionnalités différentes qui sont interdépendantes. Afin de garantir un fonctionnement aussi sûr que possible, il est fortement recommandé d'utiliser des tests. Il existe deux stratégies pour implémenter des tests. La première consiste à écrire des fonctions, puis à tester leur bon fonctionnement. La deuxième est d'écrire les tests en premier, puis d'implémenter les fonctions afin que les tests passent. Nous vous recommandons d'utiliser la **seconde** méthode.

1.1.1 Exemple

Imaginons que nous voulions écrire un algorithme qui calcule de PGCD de deux nombres. Nous commençons par réfléchir à la **signature** de la fonction. Notre fonction `pgcd` prendra en argument deux nombres entiers non-signés et retournera un entier non-signé, sa signature sera donc

```
fn pgcd(a: usize, b: usize) -> usize {
  unimplemented!()
}
```

Nous voulons à présent écrire un test dont le but est de vérifier que notre fonction a le comportement attendu (il est impossible de vérifier qu'une fonction ne possède aucun bug, on peut néanmoins tester un maximum de comportements possibles). On pourrait donc écrire des tests pour vérifier que `pgcd(3,9)=3`, `pgcd(9,3)=3`, ... En rust, il suffit d'écrire le bout de code suivant et de le placer dans le même fichier que la fonction `pgcd`

```rust
#[cfg(test)]
mod tests {
  use super::*;

  #[test]
  fn test_pgcd_3_9() {
      assert_eq!(
          pgcd(9, 12), 3
      );

      assert_eq!( // symétrie
          pgcd(9, 12), pgcd(12, 9)
      );
  }

  #[test]
  fn test_pgcd_11_14() {
      assert_eq!(
          pgcd(11, 14), 1
      );
  }

  #[test]
  fn test_pgcd_27_54() {
      assert_eq!(
          pgcd(27, 54), 27
      );
  }
}
```

Une fois ces tests écrits, nous remplissons le corps de la fonction `pgcd` ci-dessus afin de faire en sorte que tous les tests passent. Cette méthode de développement, appelée **développement piloté par les tests** (ou **test-driven development** en anglais), bien que pouvant sembler plus longue à priori, possède un grand nombre d'avantages.

1. Elle nous fait réfléchir aux tâches que nous voulons que chaque fonction accomplisse (ses fonctionnalités) **avant** de les écrire.
2. Elle nous fait réfléchir à la signature de chaque fonction.
3. Nous oblige à écrire des tests pour chaque tâche (fonctionnalité) et à traiter les cas peut-être particuliers.
4. Finalement, étant donné qu'il est relativement difficile de tester plusieurs fonctionnalités à la fois, cela nous contraint à avoir des fonctions qui effectuent un nombre limité de tâches.

Tout cela nous fait économiser beaucoup de temps sur le long terme. En commençant immédiatement à programmer sans réfléchir, on prend le risque de ne pas faire un design cohérent et à se retrouver à tout devoir reprogrammer ou pire à devoir faire des "patchs" en vitesse qui pourraient introduire des erreurs très difficilement détectables. De plus, ici, nous avons immédiatement des moyens

de tester les éventuelles régressions de notre code lorsque nous ajoutons des fonctionnalités ou faisons un refactoring.

1.2 Recommandations

Il faut tenter d'utiliser un maximum les concepts que vous avez déjà vus dans les travaux pratiques précédents.

Entre autres:

1. Modularisez votre code: organisez votre code en plusieurs fichiers/librairies lorsque cela s'y prête.
2. Séparez votre code en fonctions aussi "petites" que possible: à chaque fonctionnalité sa fonction (pensez à ne pas exagérer non plus, additionner un à un entier ne nécessite pas une fonction à part).
3. N'oubliez pas d'essayer d'avoir des codes qui sont aussi résistants aux erreurs que possible (utilisez les `Option` et `Result`), mais n'hésitez pas également à avoir des "paniques" (`panic!()`) lorsque des comportement trop "graves" ou totalement interdits pour être corrigés se produisent.
4. Commentez votre code de façon raisonnable. Il est raisonnable de documenter vos librairies (ce que fait votre librairie), il n'est pas raisonnable de commenter la fonction `fn pgcd(a: i32,b: i32) -> i32` par `// calcule le pgcd de deux entiers non-signés et retourne un entier non-signé`. De même, la ligne `let a = b + 1` n'a pas besoin de se commenter.
5. Utilisez autant que possible l'auto-documentation: vos fonctions et variables doivent avoir des noms aussi "documentants" que possible (cela vous évitera de les documenter explicitement). Lorsque l'auto-documentation n'est pas possible et que le code n'est pas suffisamment explicite (ou que vous pensez que vous utilisez une méthode non-triviale n'hésitez pas).

Toutes ces bonnes habitudes que vous prenez maintenant, vous servirons plus tard dans vos études et dans votre vie professionnelle.

Soyez proactifs. Profitez des séances de travaux pratiques pour poser un maximum de questions et n'attendez pas la veille du rendu pour nous bombarder de mail...

Vous êtes également fortement encouragés à aller au libre service pour obtenir des conseils et de l'aide pour avancer votre TP.

1.3 Rendu

Votre code source doit être déposé sur `https://cyberlearn.hes-so.ch` avant le **9.12.2018 avant 23h55**.

Il doit être rassemblé dans une archive `.zip` (ou `.tar.gz` ou `.tgz` ou `.tar.bz2` ou ...) à votre nom. Plus précisément, l'étudiant `Michel Lazeyras` mettra ses trois projets cargo, **après avoir fait un `cargo clean` dans chacun d'entre eux** dans un répertoire nommé `michel_lazeyras` qui sera lui même zippé en un fichier nommé `michel_lazeyras.zip` (une des autres extensions de votre choix). Si vous utilisez une librairie externe qui ne se télécharge pas automatiquement (par exemple la librairie de fractions dans polynômes), n'oubliez pas de l'inclure et

de spécifier son **chemin relatif** dans le fichier `Cargo.toml` (vous serez pénalisés si vos codes ne compilent pas à cause de chemins mal spécifiés).

Sous peine de sanction, vous devez respecter toutes ces spécifications. Ce travail pratique est noté. L'évaluation sera faite sur la base de votre code et d'une interrogation orale lors de laquelle vous expliquerez le travail réalisé.

2 fractions

2.1 Buts

- Mettre en œuvre la notion de traits et de généricité.
- Utiliser les fonctions récursives.
- Création de librairie.
- Gestion d'erreurs.
- Utilisation et écriture de tests unitaires.

2.2 Énoncé

Il s'agit d'écrire une librairie pour gérer des fractions. Cette librairie offrira notamment comme fonctionnalités la saisie, l'affichage, l'addition, la soustraction, la multiplication et la division de fractions (utiliser les traits `Add`, `Sub`, `Mul` et `Div`). Les fractions devront toujours être stockées sous forme irréductible. Il faudra également gérer la division par zéro qui produira une erreur.

Ensuite, il faudra écrire un programme utilisant la librairie, qui affiche le résultat d'un calcul passé en argument à la ligne de commande (comme l'addition de deux fractions).

2.3 Cahier des charges

La librairie sera constituée **au moins** de

- Une `struct Fraction` composée de 2 champs: le numérateur et le dénominateur;
- 5 fonctions:
  - Une fonction qui lit une fraction au clavier en gérant les erreurs de saisie;
  - Une fonction qui affiche une fraction à l'écran.
  - Une fonction qui rend une fraction irréductible.
  - Une fonction qui calcule le PGCD de deux nombres entiers positifs.
  - Une fonction qui met une fraction à une puissance entière et retrourne une fraction irréductible.
- 5 traits:
  - Le trait `Add` qui additionne soit deux fractions, soit un entier et une fraction et retourne une fraction irréductible.
  - Le trait `Sub` qui soustrait soit deux fractions, soit un entier et une fraction et retourne une fraction irréductible.
  - Le trait `Mul` qui multiplie soit deux fractions, soit un entier et une fraction et retourne une fraction irréductible.

    - Le trait `Div` qui divise soit deux fractions, soit un entier et une fraction et retourne une fraction irréductible (attention à la division par zéro).
    - Le trait `Neg` qui retourne le négatif d'une fraction.
- Des tests pour chacune de ces fonctions, à exécuter avec `cargo test`.
- Sur `cyberlearn` vous trouverez un fichier `lib.rs_fractions` contenant une série de tests que votre code devra passer. Vous devez implémenter les fonctions telles que ces tests passent.

Le lancement du programme avec deux fractions (ou une fraction et un entier selon l'opération) et une opération (`+`, `-`, `x`, `/`, `^`) en argument sur la ligne de commande doit afficher le résultat du calcul. Sinon on passe en argument deux nomres et PGCD et le PGCD s'affiche.

2.3.1 Exemples:

```
> cargo run 3 10 + 8 15
5 6
> cargo run 3 10 x 8 15
4 25
> cargo run 3 x 8 11
24 11
> cargo run 3 10 x 8
12 5
> cargo run 3 10 ^ 2
9 100
> cargo run 15 10 PGCD
5
```

2.4 Remarques

- Pour le calcul de la fonction PGCD, utiliser l'algorithme de division d'Euclide (sous sa forme récursive de préférence).
- Les tests s'exécutent avec la commande `cargo test`.
- Certain tests sont faits pour "paniquer". Ils sont annotés avec `#[should_panic]`.

3 polynomes

3.1 Buts

- Mettre en œuvre la notion de tableau non-contraint (`Vec`).
- Utilisation de fonctions.
- Création et utilisation de librairies.
- Gestion d'erreurs.
- Utilisation et écriture de tests unitaires.
- Utilisation des traits et généricité.

3.2 Énoncé

Il s'agit d'écrire une librairie pour la gestion de polynômes à coefficients fractionnaires. Cette librairie offrira comme fonctionnalités la saisie, l'affichage, l'addition, la soustraction, la multiplication et la division (quotient et reste) des polynômes, ainsi que l'évaluation sur une fraction. Écrire ensuite un programme qui utilise la librairie, et qui affiche le résultat d'un calcul passé en argument sur la ligne de commande (comme l'addition de deux polynômes).

3.3 Cahier des charges

La librairie sera constituée **au moins** de

- Une `struct Polynom` composée d'un champs (pourquoi pas anonyme): la représentation du polynôme sous la forme d'un tableau contenant des fractions.
- 1 fonction:
  - Une fonction qui lit un polynôme au clavier en gérant les erreurs de saisie;
- 2 méthodes:
  - Une méthode qui affiche un polynôme à l'écran.
  - Une méthode qui évalue le polynôme sur une fraction (de préférence via la méthode de Horner).
- 6 traits:
  - Le trait `Add` qui additionne deux polynômes.
  - Le trait `Sub` qui soustrait deux polynômes.
  - Le trait `Mul` qui multiplie soit deux polynômes, soit un polynôme et une fraction.
  - Le trait `Div` qui divise deux polynômes et retourne le quotient de la division.
  - Le trait `Neg` qui retourne le négatif d'un polynôme.
  - Le trait `Rem` qui le reste de la division de deux polynômes.
- Des tests pour chacune de ces fonctions, à exécuter avec `cargo test`.

Un polynôme doit toujours avoir son dernier coefficient non-nul (celui correspondant au degré du polynôme), sauf si celui-ci est une constante (un polynôme de degré zéro). Les différentes fonctions implémentées doivent le garantir.

Le programme lancé à la ligne de commande doit prendre en argument deux polynômes et une opération (`+`, `-`, `x`, `/`, `%`, e) doit afficher le résultat du calcul.

3.3.1 Exemples:

L'addition de

$$\frac{1}{2} - 2x^2 + \frac{13}{4}x^3 + \frac{7}{3}x^4$$

et

$$-\frac{3}{2} + \frac{7}{2}x + \frac{15}{6}x^2$$

doit s'écrire

```
> cargo run 1 2 0 1 2 1 13 4 7 3 + 3 2 7 2 15 7
1 1 7 2 1 7 13 4 7 3
```

Pour l'évaluation de

$$-\frac{3}{2}+\frac{7}{2}x+\frac{15}{7}x^2,$$

au point $x = 1$.

```
> cargo run 3 2 7 2 15 7 e 1 1
29 7
```

Pour le reste de la division de

$$\frac{1}{2}+\frac{5}{2}x-2x^2+\frac{13}{4}x^3+x^4$$

par

$$-\frac{3}{2}+\frac{7}{2}x+x^2$$

```
> cargo run 1 2 5 2 2 1 13 4 1 1 r 3 2 7 2 1 1
17 16 13 16
```

Attention : le polynôme est affiché en commençant par le coefficient de degré 0 jusqu'à celui du degré du polynôme.

3.4 Remarque: valeur d'un polynôme en un point

Soit $p_n(x)$ un polynôme de degré n

$$p_n(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n.$$

et x_0 un nombre. Le calcul de $p_n(x_0)$ laisse penser qu'il faut calculer toutes les puissances de x_0, les multiplier par le coefficient correspondant, et faire la somme du tout.

Si on calcule $p_n(x_0)$ en multipliant successivement x_0 par lui-même le nombre de produits nécessaire est alors de $n + (n-1) + ... + 2 + 1 = n(n+1)/2$ quantité qui croît comme n^2, avec n le degré du polynôme.

En utilisant la **méthode de Horner**, on peut grandement améliorer cette performance en faisant le calcul de la façon suivante

$$p_n(x_0) = (...((a_nx_0 + a_{n-1})x_0 + a_{n-1})x_0 + a_{n-2}...+)x_0 + a_0.$$

Le nombre de produits est alors réduit à n, de sorte que le temps de calcul d'une fonction polynomiale en un point x_0 est seulement proportionnelle au degré du polynôme.

4 reed_solomon

4.1 Buts

— Implémentation d'un code de Reed-Solomon.
— Interpolation polynomese de Lagrange.
— Utilisation de librairies.
— Gestion d'erreurs.

4.2 Énoncé

On peut résumer un code de Reed-Solomon de la manière suivante. On considère une liste, $\vec{l}$, de k octets (i.e. des nombres entre 0 et 255) à transmettre dans un canal bruité. L'encodage consiste à construire le polynôme d'interpolation $p_{k-1}(x)$ passant par la paire indice-valeur de la liste

$$p_{k-1}(0) = l_0,\ p_{k-1}(1) = l_1, ..., p_{k-1}(k-1) = l_{k-1}.$$

Puis, il faut ajouter n points de valeur

$$p_{k-1}(k),\ p_{k-1}(k+1),\ ...,\ p_{k-1}(k-1+n),$$

à la listre pour la rendre plus robuste. La liste augmentée de $k+n$ octets est ensuite transmise dans un canal bruité, c'est-à-dire qu'un certain nombres d'octets sont modifiés. A la réception de la liste bruitée de $k+n$ octets, on enlève le bruit en considérant des sous-ensembles de k éléments de cette liste et en déterminant son polynôme d'interpolation associé. On comptabilise le nombre d'apparitions d'un polynôme et on retient celui qui es majoritaire. Si le nombre d'erreurs est $\leq n/2$, alors on est certain de retrouver le polynôme d'origine et d'avoir débruité le message $p_{k-1}(0)$, ..., $p_{k-1}(k-1)$.

4.3 Cahier des charges

Il s'agit d'écrire un programme qui implémente la méthode de Reed-Solomon. Pour cela, vous devez utiliser les paquetages de gestion de polynômes à coefficients fractionnaires, créés précédemment. Dans la procédure principale, on fixe les valeurs de n et k. Ensuite, on entre k nombres entre 0 et 255 (la liste d'octets à envoyer) et on affiche le polynôme d'interpolation associé. Puis on effectue successivement les étapes d'encodage (c'est-à-dire l'ajout de n points supplémentaires), de bruitage, de débruitage et finalement de décodage telles que décrites précédemment. A chaque étape, on affiche les listes d'octets résultantes. Pour le débruitage, il faut aussi afficher le tableau des polynômes d'interpolation associés aux sous-ensembles de k éléments, ainsi que le nombre d'occurrences de ces polynômes.

En plus de la procédure principale, vous devez au moins écrire :

- une fonction (récursive) produit qui retourne un polynôme égal au produit $(x-a_1)\cdot(x-a_2)\cdot ...\cdot(x-a_n)$;
- une fonction `interpolate` qui calcule le polynôme d'interpolation de Lagrange d'un ensemble de points.

Le programme aura deux modes de fonctionnement. S'il est lancé sans argument sur la ligne de commande, il s'exécute en interactif. Par contre, s'il est lancé avec une liste d'entiers x_1, y_1, x_2, y_2,..., x_n, y_n en argument sur la ligne de commande, comme par exemple : `cargo run 1 4 2 7 8 9 1 2` alors le programme *affichera uniquement* le polynôme d'interpolation par les points (x_1, y_1), (x_2, y_2),..., (x_n, y_n) dans le format du package de gestion des polynômes.

4.4 Références

- Reed-Solomon
- Polynôme de Lagrange