# Projet Algorithmiques avancés L2-L3 SdN semestre 2

## Cadre

Les permutations sont des objets communément utilisés en mathématiques. Formellement, il s’agit d’une bijection d’un ensemble fini dans lui-même. Pour le travail, le nombre d’éléments de sera noté et sera assimilé à l’ensemble des entiers compris[[1]](#footnote-1) entre et . L’objectif est de définir une bibliothèque C permettant de gérer une type permutation. Afin de ne pas mélanger une permutation sur des ensembles de tailles différentes, nous procéderons de la manière suivante.

### Bref survol & esprit général de l’approche

La bibliothèque sur les permutations devra être initialisée avant de pouvoir allouer les ressources nécessaires à la moindre permutation. Lors de cette initialisation la taille[[2]](#footnote-2) de **toutes** les permutations est définie et ne pourra pas évoluer durant l’exécution du programme. Le travail est découpé en trois parties : deux parties de développements et une partie rapport.

Les deux fichiers d’entête vous imposent choix d’implantation :

* Le type t\_boolean servira à manipuler des valeurs pouvant servir de condition en C.
  + 0 : FAUX;
  + 1 : VRAI.

De nombreuses fonctions de la bibliothèque sur les permutations retournent un t\_boolean :

* + FAUX : la fonction s'est mal déroulée (les conditions d'utilisation ne sont pas remplies) ;
  + VRAI : la fonction s'est bien déroulée (les Conditions d'utilisation sont remplies).

Certaines fonctions sont des prédicats qui retourne VRAI uniquement dans le cas où les conditions d’utilisation sont remplies **et** si le prédicat est vérifié.

* Le type t\_permutation est un pointeur sur **des** valeurs de type int. Ainsi les prototypes des fonctions ne signalent pas si le contenu une permutation est modifié par la fonction.
* Les prototypes des fonctions.
  + Vous êtes autorisés à ajouter des fonctions auxiliaires dans les fichiers .c afin de vous faciliter le développement.
  + Vous pouvez également discuter **dans le rapport** :
    - La pertinence des prototypes ;
    - L’ajout de fonctionnalités supplémentaires.

Ce qui permettra à votre professeur de faire évoluer le sujet pour des **prochaines** promotions.

* + Quoiqu’il en soit, vous ne pouvez pas supprimer des prototypes des deux fichiers d’entête. En effet, Le correcteur appliquera des routines[[3]](#footnote-3) pour valider votre développement.

Les fichiers d’entête vous proposent des choix d’implantation :

* Vous remarquerez[[4]](#footnote-4) que certaines fonctions facilitent le développement des autres fonctions. Bien que la réutilisation de ces programmes ne soit pas obligatoires. Vous êtes **grandement** incités à exploiter vos premiers développements. Sachez que la pénalité viendra d’elle-même car certaines fonctions deviennent extrêmement pénibles à implanter si vous n’exploitez pas les travaux préliminaires.

### Plan du document

L’énoncé s’articule en trois sections : La section **Notions sur les permutations** récapitule des connaissances sur les permutations. La section **Ressource** présente tous les fichiers qui sont mis à votre disposition. La section **Travail à réaliser** présente ce qui est attendu pour le projet. La section **Évaluation** présente les modalités pour le calcul de la note de votre projet ; Elle peut vous guider en vous indiquant des pistes ou des points d’amorce de votre travail.

## Notions sur les permutations

### Représentation

Comme cela a été évoqué en introduction, une permutation est une bijection de dans lui-même. Une permutation peut être représentée mathématiquement par une fonction bijective. Contrairement au fonction numérique dont vous avez l’habitude en analyse, une fonction sur peut être **implantée** par *son tableau de valeurs*. Ainsi, on fournit pour chaque indice compris entre et son image, noté , qui est de type int.

Vous avez appris qu’une fonction est bijective si elle est à la fois *injective* et *surjective*. Dans notre cas, cette contrainte se traduit par le fait que les valeurs du tableau qui représente sont toutes différentes.

### Réciproque

Vous savez que si est une bijection alors il existe une fonction, notée tel que et . Cette fonction est appelée *fonction réciproque* de .

Vous devrez trouver par **vous-même[[5]](#footnote-5)** un algorithme permettant de construire la réciproque d’une permutation.

### Suivant

Par un exemple avec N = 4. On peut générer toutes[[6]](#footnote-6) les permutations :

[ 0 1 2 3 ]

[ 0 1 3 2 ]

[ 0 2 1 3 ]

[ 0 2 3 1 ]

[ 0 3 1 2 ]

[ 0 3 2 1 ]

[ 1 0 2 3 ]

[ 1 0 3 2 ]

[ 1 2 0 3 ]

[ 1 2 3 0 ]

[ 1 3 0 2 ]

[ 1 3 2 0 ]

[ 2 0 1 3 ]

[ 2 0 3 1 ]

[ 2 1 0 3 ]

[ 2 1 3 0 ]

[ 2 3 0 1 ]

[ 2 3 1 0 ]

[ 3 0 1 2 ]

[ 3 0 2 1 ]

[ 3 1 0 2 ]

[ 3 1 2 0 ]

[ 3 2 0 1 ]

[ 3 2 1 0 ]

L’objectif des fonctions suivante\_permutation et copier\_suivante\_permutation doit permettre de passer d’une permutation à la permutation suivante. Si la permutation n’a pas de permutation suivante alors ces fonctions l’indiquent en retournant FAUX.

Ce point-ci est assez technique, voici quelques éléments pour vous aider à implanter un algorithme :

1. Repérer la plus grande tranche décroissante à la fin de la permutation .
   1. On notera l’indice de l’élément situé **juste avant** le début de cette tranche ;
   2. Si vaut alors la permutation n’a pas de suivante (i.e. est la dernière permutation de la liste).
2. Dans cette tranche repérer le plus petit élément supérieur à
   1. On notera l’indice de cet élément ;
3. Échanger les éléments situés aux indices et .
4. Inverser l’ordre des éléments situés après .

La Figure 1 présente la démarche sous la forme d’un petit story-board (à lire colonne de gauche puis la colonne de droite).

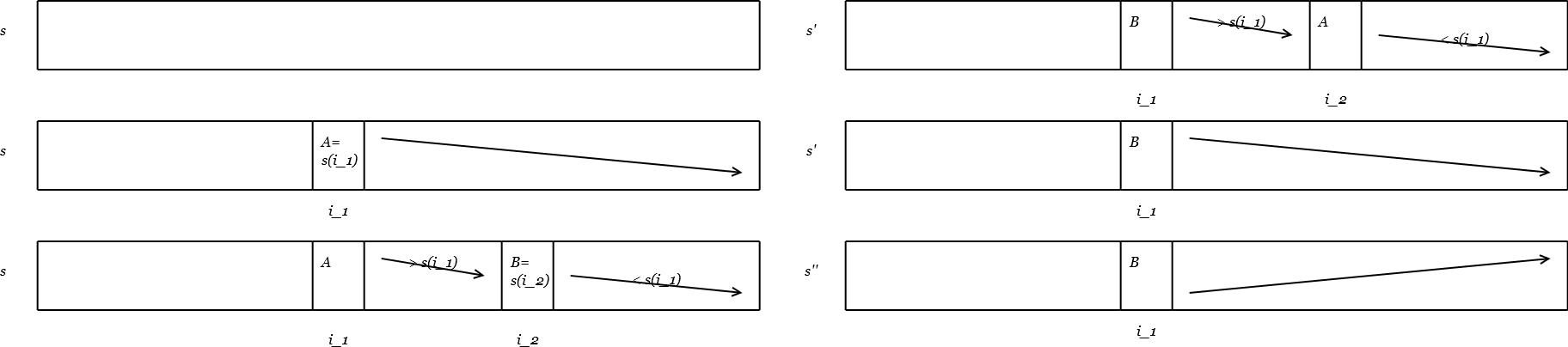


Figure -illustration de l'algorithme pour passer d'une permutation à sa permutation suivante.

### Points-fixes

La notion de point-fixe[[7]](#footnote-7) n’est pas spécifique aux permutations : Elle existe dès l’étude de fonctions d’un ensemble dans lui-même, notamment en analyse numérique.

**Définition** : soit un ensemble et une fonction de dans . On appelle « *point-fixe* » de tout élément vérifiant .

Dans le cadre des permutations, on peut notamment étudier le nombre de points fixes d’une permutation. Certaines possèdent de nombreux points fixes, comme l’identité [ 0 1 2 3 ]. D’autres permutations n’en possèdent pas, comme la permutation [ 2 3 1 0 ] par exemple. On se demande quel est le nombre moyen de points fixes sur l’ensemble des permutations.

Pour cela, on peut commencer par une approche expérimentale. Dans un premier temps, on compte le nombre de points fixes pour toutes les permutations pour un ordre[[8]](#footnote-8) donné. Puis on recommence en faisant varier cet ordre. On espère ainsi voir apparaitre une propriété remarquable et pourvoir conjecturer un résultat.

Dans un second temps, le mathématicien[[9]](#footnote-9) cherche à prouver cette conjecture. Vous pourrez le faire dans le rapport en suivant la démarche suivante :

1. Montrer que la fonction code\_point\_fixe est injective.
2. Montrer que la fonction decode\_point\_fixe est injective.
3. Prouver que les fonctions code\_point\_fixe et decode\_point\_fixe réalisent des travaux réciproque l'un de l'autre.
4. En déduire le nombre de codage des points fixes.
5. En déduire le nombre de points fixes.

## Ressources

Vous disposerez des fichiers :

* p\_base.h : le fichier d’entête pour les routines non spécifique à notre objectif ;
* p\_permutation.h  : le fichier d’entête pour les routines spécifiques à notre objectif ;
* test\_permutation.c : un brouillon de programme qui permet de faire des tests sur vos programmes. **Attention** : la routine de test pour l’évaluation sera plus riche que celui-ci ;
* makefile : un fichier qui réalise les dépendances pour une compilation séparée de test\_permutaion.exe.

## Travail à réaliser

### Rendu

Vous rendrez votre travail sur icampus sous la forme d’une archive (zip ou rar) avant la date spécifiée sur la plateforme. Votre archive devra contenir :

* Vos programmes C : p\_base.c et p\_permutation.c ;
* Un rapport présentant le projet (cf Kit de survie du rapport).

Votre rendu doit faire mention des deux membres du binôme.

**Attention** : si les **deux** membres du groupe font des rendus séparés. Le correcteur en corrigera un seul (au hasard). Pour être bien clair : Un ajout de dernière minute par l’un des membres du groupe pourra être corrigé et bénéficier au deux membres du groupe ou pourra être ignorer pour les deux selon l’exemplaire pioché par le correcteur.

**Attention** : les rendus (dépôt sur icampus) en retard seront pénalisés en appliquant la procédure stipulée par le règlement.

**Remarque** : Vous pouvez échanger entre vous sur vos différentes approches et les difficultés que vous allez surmonter mais je vous demande de ne pas échanger de code : uniquement des concepts, des schémas et notes manuscrites.

### Développement

Vous implanter les bibliothèques :

* p\_base (fichier p\_base.c).
* p\_permutation (fichier p\_permutation.c). La bibliothèque

### Production p\_base.c

Cette bibliothèque contient les éléments non spécifiques à notre application.

* Type t\_boolean servant essentiellement de valeur de retour indiquant la bonne utilisation d’une fonction.
* Une fonction pour générer des entiers aléatoires entre deux bornes données.
* Des fonctions sur les tableaux d’entiers sans contrôle particulier : chaque fonction s’exécute en faisant ce pourquoi elle a été prévue même si les paramètres sont erronés. Dans ce cas, la fonction fera n’importe quoi et pourquoi une erreur de segmentation.
  + Une procédure pour remplir un tableau avec des valeurs permettant d’assimiler ce tableau à la fonction identité de l’intervalle du type ;
  + Une procédure pour échanger deux éléments d’un tableau ;
  + Une procédure pour inverser l’ordre des éléments d’un tableau ;
  + Une procédure pour copier un tableau dans un autre ;
  + Une procédure pour copier un tableau dans un autre en inversant l’ordre des éléments ;
  + Une procédure pour indiquer si un tableau est inférieur à un autre tableau de même taille pour l’ordre lexicographique.

### Production p\_permutation.c

Cette bibliothèque encapsule un certain nombre de savoir-faire sur les permutations. Elle ajoute des vérifications pour éviter que le programmeur fasse des erreurs trop grossières[[10]](#footnote-10). Elle contient :

* La définition du type t\_permutation ;
* La définition d’une constante NON\_INITIALISER\_PERMUTATION pour *inciter* le programmeur à utiliser cette valeur lors de la déclaration des variables de type t\_permutation ;
* Une fonction pour initialiser la bibliothèque ;
* Un moyen d’indiquer si la bibliothèque a été initialisée ;
* Un moyen d’indiquer au programme la taille des permutations. Ces deux dernières fonctions pourraient servir si le programme possède deux parties qui ont été développées séparément. La première initialisant la bibliothèque et la seconde exploitant les permutations ;
* Une fonction pour initialiser des permutations. On s’attend à ce que le programmeur initialise la *bibliothèque* avant d’initialiser des permutations ;
* Une fonction pour initialiser une permutation, allouant la mémoire pour stocker les valeurs d’une permutation ;
* Une fonction pour finaliser une permutation, libérant la mémoire réservée par la fonction précédente ;
* Une procédure pour fixer la valeur d’une permutation à l’identité ;
* Une procédure pour copier une permutation dans une autre permutation ;
* Une procédure pour afficher une permutation. Vous ferez attention au format afin que le programme d’évaluation puisse vous noter ;
* Un prédicat indice valide qui retourne vrai si son paramètre appartient à l’intervalle d’entiers , où est l’ordre des permutations ;
* Une procédure pour mélanger les valeurs d’une permutation ;
* Une fonction pour obtenir dans le second paramètre la réciproque du premier paramètre ;
* Un prédicat permettant de vérifier si un tableau de nombre représente effectivement une permutation (i.e. tous les valeurs entre et sans doublon) ;
* Un prédicat encapsulant la relation d’ordre lexicographique sur les permutations ;
* Un prédicat permettant de savoir si une valeur est un point fixe pour une permutation ;
* Une fonction permettant de connaitre le nombre de points fixes d’une permutation ;
* Une procédure pour changer une permutation en sa suivante ;
* Une procédure pour obtenir dans le second paramètre la permutation suivante de son premier paramètre ;
* Une fonction pour coder un point fixe en une permutation ;
* Une fonction pour réaliser le décodage inverse de la fonction précédente ;

### Rapport

En informatique comme dans de nombreux domaines, tous les travaux sont accompagnés par un document qui relate le travail effectué. Ce document se nomme *rapport*. Il est nécessaire à l’évaluateur pour estimer la pertinence du travail réalisé. Comme vous le verrez dans la partie suivante la partie compte pour une partie **importante** dans la note de projet. Vous aurez un document annexe présentant un **kit de survie** d’un rapport.

## Évaluation & barème

Le tableau suivant présente une grille d’évaluation. Les cases grisées indiquent des sommes partielles des coefficients de la partie. Ce barème a pour but de vous aider à amorcer le travail et identifiant des points clés du projet.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Projet | Rendu conforme | | Makefile | 0,4 |
| Archive | 0,2 |
| Compilation | 0,4 |
| **1** | |
| Rapport | Document | Langue | 0,5 |
| Traitement de texte | 0,5 |
| **1** | |
| Introduction | Problème | 1 |
| Équipe | 0,5 |
| Méthode de travail | 0,5 |
| **2** | |
| Solution | Technologie | 1,5 |
| Difficultés surmontées | 0,5 |
| **2** | |
| Conclusion | Expérience | 0,5 |
| Prolongation | 0,5 |
| **1** | |
| **6** | | |
| Développement | p\_base | aleatoire\_entre\_a\_et\_b | 0,3 |
| initialiser\_tableau | 0,8 |
| echanger\_tableau | 0,3 |
| croiser\_tableau | 0,9 |
| copie\_tableau | 0,8 |
| copie\_croisement\_tableau | 0,9 |
| **4** | |
| p\_permutation | initialiser\_p\_permutation | 0,3 |
| donner\_taille\_permutation | 0,3 |
| est\_p\_permutation\_initialise | 0,3 |
| initialiser\_permutation | 1 |
| finaliser\_permutation | 0,6 |
| copier\_permutation | 0,3 |
| est\_indice\_valide | 0,3 |
| melanger\_permutation | 0,6 |
| reciproque\_permutation | 1 |
| est\_permutation\_valide | 0,8 |
| est\_inferieur\_strict | 0,3 |
| est\_point\_fixe | 0,3 |
| nb\_points\_fixes | 0,4 |
| suivante\_permutation | 1 |
| code\_point\_fixe | 0,9 |
| decode\_point\_fixe | 0,6 |
| **9** | |
| **13** | | |
| **20** | | | |

1. Dans le texte, les deux bornes, et sont incluses. Cependant dans vos programmes vous pourrez travaillé avec la borne inférieur inclus et la borne supérieur exclu, pour simplifier l’écriture de vos programmes en C. [↑](#footnote-ref-1)
2. Les matheux emploient plus volontiers de terme « d’ordre » mais il me semble taille parle plus à des informaticiens. [↑](#footnote-ref-2)
3. Un programme qui compilera votre bibliothèque et l’exécutera pour obtenir une partie de la note. [↑](#footnote-ref-3)
4. Espérons-le. [↑](#footnote-ref-4)
5. Le **sujet** ne donne pas d’indice mais vous pouvez discuter dans la promotion et avec le professeur. [↑](#footnote-ref-5)
6. 4! Soit 24 permutations pour notre exemple. [↑](#footnote-ref-6)
7. Il ne s’agit pas du petit chien d’Obélix. [↑](#footnote-ref-7)
8. On rappelle que le mot *ordre* est le terme savant pour dire « taille du tableau ». [↑](#footnote-ref-8)
9. Un gars paranoïaque qui croit uniquement les preuves solides. [↑](#footnote-ref-9)
10. Cependant, le contrôle suggéré ici n’est pas absolu. On peut encore améliorer celui-ci mais il restera toujours des failles. [↑](#footnote-ref-10)