
TP3 : Génération et dénombrement.**Preliminaires.**

Téléchargez depuis le site l'archive **TP3.zip** et extrayez les fichiers dans votre dossier personnel puis vérifiez la compilation du fichier **TP3.cpp**.

Exercice 1 (Programmation récursive / dynamique).

Dans cet exercice, on explore les liens entre dénombrement et récursivité et deux stratégies de programmation : **programmation récursive** et **programmation dynamique**.

Commençons par un exemple. Nous avons implanté deux fonctions calculant les **nombre de Fibonacci**. Nous utilisons une définition un peu différente de celle usuelle. Soit f_n la suite définie telle que

$$\begin{aligned} f_n &= 0 & \text{si } n < 0 \\ f_0 &= 1 \\ f_n &= f_{n-1} + f_{n-2} & \text{si } n \geq 1. \end{aligned}$$

Vous remarquerez qu'on a défini la suite pour tous les entiers positifs et négatifs et que le premier non-nul est f_0 (dans la définition usuelle, c'est f_1).

Versión récursives. Observez la fonction **fibonacciRec** du fichier. Elle utilise un principe *récursif* : la fonction s'appelle elle-même.

Versión dynamique. Le problème de cette implantation récursive est que l'on effectue plusieurs fois les mêmes calculs. Par exemple, pour calculer f_4 , je calcule f_3 et f_2 . Mais pour calculer f_3 , je dois aussi calculer f_2 . Tel que le programme est écrit, on n'a pas de *mémoire* des calculs précédents. La conséquence est que l'on effectue un nombre de calcul de l'ordre de 2^n pour f_n .

On propose donc une nouvelle approche par **programmation dynamique** : on crée un tableau de taille n et on calcule les valeurs successives de la fonction que l'on stocke dans le tableau ($t[i]$ doit contenir f_i). Pour chaque case du tableau, on calcule la valeur en utilisant les cases précédentes. A la fin, la dernière case du tableau nous donne f_n .



Par exemple, pour calculer f_5 , on doit créer le tableau suivant :

1	1	2	3	5	8
---	---	---	---	---	---

- (1) Complétez les fonctions **gRec** et **gDyn** dont la description est donnée en commentaire en vous inspirant des deux implantations de la fonction fibonacci et lancez les tests correspondants.
- (2) **Fibonacci et les murs de brique.** Observez le phénomène suivant : je possède des briques de longueur 1 et 2, le nombre de différents alignements de longueur n que je peux réaliser est donné par la suite de Fibonacci. En effet :

Murs de taille 0 : 1 (le mur vide)

Murs de taille 1 : 1 

Murs de taille 2 : 2  

Murs de taille 3 : 3   

Murs de taille 4 : 5     

Pourquoi ? Observons les alignements de taille 4.

- les **3 premiers** terminent par une brique de taille 1. Ils correspondent aux 3 murs de taille 3 auxquels on a rajouté une brique de taille 1 à la fin.



- les **2 derniers** terminent par une brique de taille 2. Ils correspondent aux 2 murs de taille 2 auxquels on a rajouté une brique de taille 2 à la fin.



Sur le même principe, mes briques de longueur 2 ont à présent 2 couleurs différentes : rouges ou bleues (les briques de longueur 1 sont toutes blanches) et je me demande combien d'alignements différents je peux réaliser. Voilà par exemple, les alignements possibles jusqu'en taille 3 (on a noté 1 pour les briques de taille 1, *R* pour les briques rouges de taille 2 et *B* pour les briques bleues de taille 2).

Murs de taille 0 : 1 (le mur vide)

Murs de taille 1 : 1 - 1 -

Murs de taille 2 : 3 - 11 - *R* - *B*

Murs de taille 3 : 5 - 111 - *R*1 - *B*1 - 1*R* - 1*B*

- (a) Pour faire la liste des murs de taille 4 :

— on prend les murs de taille 3 où je rajouter une brique 1 : 1111 - *R*11 - *B*11 - 1*R*1 - 1*B*1

— et les murs de taille 2 où je rajoute une brique *R* : 11*R* - *R**R* - *B**R*

— et les murs de taille 2 où je rajoute une brique *B* : 11*B* - *R**B* - *B**B*

On trouve donc **11 murs de taille 4**. Faites de même avec les murs de taille 5, combien en trouvez-vous ?

- (b) Exprimez le nombre de murs par une formule récursive.

- (c) Implantez les deux fonctions `rougeBleuRec` et `rougeBleuDyn` qui implante cette règle selon les stratégies récursives et dynamiques vues précédemment et vérifier que les tests passent.

- (3) **Scores de Rugby.** Au rugby, différentes situations permettent de marquer différents nombres de points :

- Marquer un essai donne 5 points, et donne droit à tenter une transformation, qui apporte 2 points supplémentaires si elle est réussie.
- Un tir entre les poteaux donne 3 points.
- Une pénalité donne 3 points si elle est réussie.

On veut compter le nombre de déroulement de matchs possibles qui peuvent amener à un score donné. On appelle un déroulement de match une séquence de points effectuée dans un certain ordre. Par exemple, pour obtenir le score 10, on compte 5 séquences :

- Un tir puis un essai transformé ;
- Une pénalité puis un essai transformé ;
- Deux essais non transformés consécutifs ;

- Un essai transformé puis un tir ;
 - Un essai transformé puis une pénalité.
- (a) Exprimez la formule mathématique récursive qui répond à la question (de façon similaire à Fibonacci). On peut réfléchir de cette façon : comment, en utilisant une séquence plus petite, puis-je obtenir une séquence terminant par un essai transformé ? Un essai non transformé ? Un tir ? Une pénalité ?
- (b) Utilisez cette formule pour implanter les deux fonctions `nbSeqRugbyRec` et `nbSeqRugbyDyn` qui calculent ce nombre selon les stratégies récursives et dynamiques.

Exercice 2 (Génération exhaustive).

Dans l'exercice précédent, nous nous sommes contentés de compter le nombre de solutions possibles. À présent, on voudrait fabriquer explicitement l'ensemble des solutions à un problème récursif donné.

- (1) Observez la fonction `afficheCouples` qui affiche tous les couples d'entiers (i, j) pour i et j entre 0 et n . Implantez de façon similaire la fonction `afficheTriplets` qui affiche les triplets de nombres (i, j, k) .

Que faire à présent si l'on souhaite afficher tous les k -uplets d'entiers ? C'est-à-dire tous les vecteurs de dimension k dont les valeurs sont comprises entre 0 et n . C'est la question que nous allons résoudre à présent. Par ailleurs, nous ne nous contenterons pas de les afficher mais de *créer* réellement les objets `vector` correspondants. Pour cela, on utilise une `Collection`, c'est-à-dire un ensemble de vecteur d'entiers.

```
typedef vector<vector<int>> Collection ;
```

Deux fonctions vous sont données : `afficheVecteur` et `afficheCollection`.

- (2) Observez les trois fonctions `genere0Uplets`, `genere1Uplets`, `genere2Uplets` qui fabriquent les collections de k -uplets pour $k = 0, 1$ et 2 respectivement. Des exemples d'utilisation sont donnés dans la fonction `main`. Remarquez par exemple que la fonction `genere2Uplets` appelle `genere1Uplets` pour créer les vecteurs de taille 1 auxquels on rajoute toutes les valeurs possibles.
- (3) En vous inspirant de ces fonctions, implantez la fonction `genere3Uplets` et lancez les tests correspondants.
- (4) Le but est maintenant d'implanter une fonction générique `generekUplets` qui prend en paramètre un entier k (la taille du vecteur) et un entier n (la valeur maximale des composantes du vecteurs). Cette fonction sera **récursive** et se basera sur le principe des fonctions de la question précédente. Il vous faut traiter 3 cas :
- $k < 0$: collection vide
 - $k = 0$ collection contenant le vecteur vide (celle de `genere0Uplets`)
 - $k > 0$ collection construite à partir de $k - 1$.

Implantez la fonction `generekUplets` et lancez les tests correspondants.

Exercice 3 (Retour au Rugby).

On revient au problème des matchs de rugby, mais cette fois on s'intéresse aux configurations de matchs *sans tenir compte de l'ordre* des points marqués. Par exemple, pour le score de 10, on a maintenant 3 configuration possibles :

- 1 essai transformé, 0 essai non transformé, 1 tir, 0 pénalité ;
- 1 essai transformé, 0 essai non transformé, 0 tir, 1 pénalité ;
- 0 essai transformé, 2 essais non transformés, 0 tir, 0 pénalité ;

Une telle configuration peut être interprétée comme un **vecteur de taille 4** dont la première valeur correspond au nombre d'essais transformés, la seconde au nombre d'essais

non transformés, la troisième au nombre de tirs et la dernière au nombre de pénalités. Par exemple, les 3 configurations donnant le score 10 sont : $(1, 0, 1, 0)$, $(1, 0, 0, 1)$ et $(0, 2, 0, 0)$.

- (1) Implantez la fonction `scoreRugby` qui calcule le score associé à un vecteur. La fonction pourra accepter des vecteurs de toute taille : si la taille est inférieure à 4, seules les valeurs présentes seront prises en compte, si la taille est supérieure à 4, les valeurs en plus seront ignorées.
- (2) Implantez la fonction `matchesRugby` qui retourne la collection des vecteurs de taille 4 correspondant à un score donné. Pour cela, on utilisera la fonction `genereKUpsets` écrite précédemment et on testera chacun des vecteurs obtenu pour voir si le score est le bon.
- (3) Le problème de la fonction précédente est qu'elle est très coûteuse : en effet il y a de très nombreux vecteurs de taille 4 et très peu donnent le bon score. Par ailleurs, il est parfois inutile d'engendrer *tous* les vecteurs de taille 4. Par exemple, si mon vecteur de taille 3 commence par $(3, 2, 2)$, je sais que le score sera minimum de $3 \times 7 + 2 \times 5 + 2 \times 3 = 37$. On se propose donc d'écrire la fonction `matchesRugbyInf` qui **n'utilisera PAS** la fonction `genereKUpsets` mais utilisera une méthode récursive similaire pour engendrer les vecteurs d'une taille donnée k dont le score est inférieur ou égal à n .
- (4) Enfin, implantez la fonction `matchesRugby2` qui utilise `matchesRugbyInf` pour donner un résultat similaire à `matchesRugby`. Vous remarquerez que vous pouvez calculer pour des valeurs beaucoup plus grandes qu'avec la fonction initiale.

Exercice 4 (Aller plus loin : retour aux briques ♣).

- (1) Implantez la fonction `mursFibonacci` qui retourne la collection des vecteurs dont les valeurs ne sont que des 1 et des 2 et dont la somme fait n (la taille est quelconque).
- (2) En utilisant une méthode (dynamique ou récursive) similaire à l'exercice 1, implantez la fonction `nbMursToutesBriques` qui retourne le nombre d'alignements de brique de taille n avec des briques de longueur quelconque. Au vu du résultat, êtes-vous capable de conjecturer une formule directe ? Pouvez-vous prouver ce résultat directement ?
- (3) On se donne à présent la règle suivante pour construire notre alignement : on a le droit à toutes les tailles de briques mais deux briques consécutives ne peuvent pas avoir la même parité (une brique paire est suivie par une impaire et vis-versa). Implantez la fonction `nbMursPairsImpairs` qui calcule le nombre d'alignement selon cette règle.

Aide : séparez votre calcul en deux : alignement terminant par une brique paire et alignement terminant par une brique impaire.

- (4) Pour les questions précédentes, implantez des fonctions qui engendrent la collection de vecteurs correspondants et écrivez vos propres fonctions de tests (le nombre de vecteurs engendrés doit être le même que le nombre calculé).

Exercice 5 (Aller encore plus loin ♣♣).

Reprenez la structure de relation binaire vue à la séance précédente et créez une fonction pour engendrer l'ensemble des relations binaires d'une taille donnée. Utilisez cette fonction pour compter la proportion de relation réflexive, symétrique, anti-symétrique et transitive.