Projets d'informatique

L1.2

2010-2011

Cité Descartes, 5 Boulevard Descartes, Champs sur Marne,
77454 Marne La Vallée CEDEX 2

Présentation des projets d'informatique de L1.2

Le contrôle des connaissances en module informatique comprend la réalisation d'un projet de programmation. Ce projet doit être réalisé en **binômes**. Le travail demandé consiste à écrire le programme et à rédiger un rapport de quelques pages décrivant le mode d'utilisation et les particularités des méthodes utilisées.

Ce rapport ne **doit pas** consister en un simple commentaire de la construction du programme mais doit décrire et justifier l'algorithme et les choix de structures de données. Il devra comprendre également des exemples d'utilisation avec d'éventuelles indications de performances.

Le programme sera écrit en C et fonctionnera sur le machine de l'université. Les points suivants feront l'objet d'une attention particulière lors de la réalisation du projet :

- **modularité :** les tâches indépendantes doivent être exécutées par des fonctions ;
- **utilisation des variables :** on utilisera autant que possible des variables locales et on fera le choix de noms pertinents pour les identificateurs et les fonctions ;
- **commentaires :** le programme devra être commenté en donnant les indications nécessaires à sa lisibilité.

Les sources du programme doivent être jointes au dossier. Le projet donnera lieu à une soutenance lors de la dernière séance de T.P.. Les dossiers doivent être remis avant le début de la soutenance. Le sujet sera choisi parmi les sujets décrits dans les pages qui suivent. Exceptionnellement, un sujet personnel peut-être proposé au chargé de T.D. à condition qu'un dossier de forme analogue à celle des propositions qui suivent lui soit remis et que le sujet soit accepté.

Remarques : En cas d'incompréhension d'un énoncé, envoyez un email à Marc Zipstein (zipstein@univ-mlv.fr) décrivant **précisément** votre problème.

Bon travail

Listes des sujets de projet

1. Mini-éditeur de textes
2. Mini LOGO ou Jeu de la tortue
3. Isola
4. Calcul du polynôme caractéristique d'une matrice par la méthode de Faddeev
5. Manipulation de polynômes
6. Cryptage du sac à dos
7. Gestion de droits d'accès
8. Fermeture transitive d'un graphe
9. Variations sur la commande wc
10. Puissance 4
11. Répertoire téléphonique
12. Master Mind graphique
13. Entiers en précision arbitraire
14. Recherche exacte et approchée de mots dans un texte
15. Le jeu de la vie
16. Représentation d'un échiquier
17. Billard graphique
18. Gestion de listes
19. Bataille navale
20. Réversi
21. Représentation d'un labyrinthe
22. Trois en ligne

Mini-éditeur de textes

sujet proposé par Marc Zipstein

Le but du projet est la réalisation d'un mini-éditeur de textes. Cet éditeur sera du type ligne, c'est-à-dire que l'utilisateur n'a accès qu'à une seule ligne à la fois. Pour un grand nombre d'opérations, il faudra donc préciser sur quelle ligne on veut travailler.

La taille d'une ligne sera limitée à 80 caractères, la taille d'un texte sera limitée à 20 lignes. Les différentes fonctionnalités de l'éditeur seront appelées par un menu. On ne travaillera que sur un texte à la fois.

Les différentes fonctionnalités demandées seront appelées, à l'aide d'un menu, par les commandes suivantes :

- saisie d'un nouveau texte → `E`
- affichage d'une ligne → `a num`
- affichage du texte → `A`
- affichage du texte avec numéro de ligne → `N`
- suppression d'une ligne → `d num`
- insertion en fin de texte → `I`
- insertion avant une ligne → `i num`
- modification d'une ligne → `m ligne`
- affichage de toutes les lignes (avec leur numéro) contenant un mot donné → `f mot`
- remplacement dans tout le texte d'un mot par un autre → `s ancien nouveau`.

On pourra considérer les lignes indépendantes : une ligne trop longue est refusée plutôt que décaler le reste du texte ! Il n'est donc pas nécessaire de gérer les problèmes de césure. On pourra également utiliser la bibliothèque graphique pour afficher le texte et les numéros de lignes ou permettre une interface plus agréable. Tout apport de nouvelles fonctionnalités est bienvenu.

Mini LOGO ou Jeu de la tortue

Sujet proposé par Pierre Vinant

Description

LOGO est un langage informatique français (Cocorico !) destiné à faire apprendre l'informatique à des enfants. Au départ, un petit robot appelé la tortue, était relié à l'ordinateur. Cette tortue pouvait, grâce à des roulettes, se déplacer sur une grande feuille de papier posée sur le sol. Elle tenait un crayon qu'elle pouvait éventuellement poser sur la feuille, ce qui fait qu'en se déplaçant, elle pouvait laisser une trace sur la feuille. Les programmes consistaient donc à donner des ordres à cette tortue pour lui faire faire des dessins.

Par la suite, le robot tortue, quelque peu encombrant, a été remplacé par un petit triangle sur l'écran de l'ordinateur. Mais le principe reste le même, à savoir, faire un dessin en déplaçant ce petit triangle.

Le projet

Le projet consiste à réaliser un mini LOGO. C'est à dire un programme qui va saisir des ordres (avance de 10 : **AV 10**, baisse le crayon : **BC**, Tourne à droite de 90^0 : **TD 90**, ...) et qui va déplacer en conséquence la tortue (i.e. le triangle à l'écran). Quand le crayon sera baissé, le déplacement de la tortue laissera une trace : le dessin. Mais attention, si le crayon est levé, la tortue ne laissera aucune trace en se déplaçant.

Voici les ordres auxquels la tortue doit savoir réagir. Dans toute la suite, le caractère **N** représente un nombre entier positif quelconque. C'est l'argument de la commande.

- **BC** : la tortue Baisse son Crayon (elle le pose sur le papier) ;
- **LC** : elle Lève son Crayon. Elle ne laisse donc plus de trace pour les instructions suivantes jusqu'à la lecture de **BC** ;
- **AV N** : elle AVance de N unités dans la direction où elle regarde ;
- **RE N** : elle REcule de N unités ;
- **TD N** : elle Tourne à Droite de N degrés autour de son crayon ;
- **TG N** : elle Tourne à Gauche de N degrés autour de son crayon ;
- **CE** : elle retourne au CEntre de l'écran et regarde vers le haut. Mais attention, si le crayon est baissé, elle laissera une trace en se replaçant ;
- **CC N** : elle Change la Couleur de son crayon. Si N = 0, elle prend la couleur noire et si N = 1, elle prend la couleur blanche. Cette commande est très utile pour effacer des traits quand on s'est trompé en faisant le dessin ;
- **EF** : pour EFfacer complètement l'écran. La tortue reste où elle est ;

- FI : pour quitter votre programme (FIn).

Initialement, la tortue se trouve au centre de l'écran et elle regarde vers le haut. Son crayon est baissé et il a la couleur noire (*cf.* Fig1).

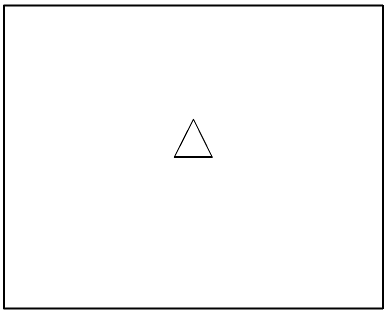


Fig 1

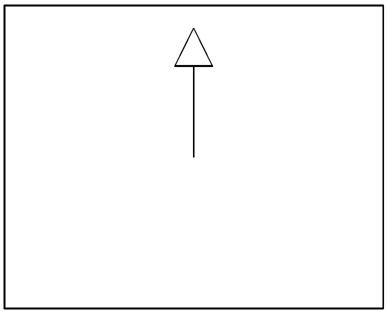


Fig 2

Attention, si vous représentez la tortue par un triangle équilatéral, on ne verra pas de quel côté elle regarde. Le triangle isocèle est le plus adapté. La tête de la tortue étant du côté de l'angle différent. Le crayon se trouve au centre de la base du triangle.

Après chaque ordre donné à la tortue, on doit en voir les conséquences sur le dessin. Les figures 2, 3 et 4 représentent respectivement ce que l'on doit voir après les commandes : **AV 40**, **TD 135** et **AV 30**.

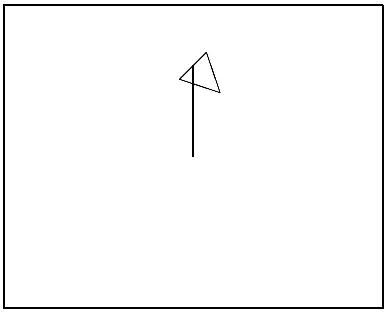


Fig 3

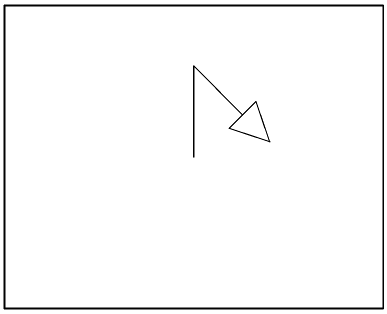


Fig 4

Améliorations possibles

Toute amélioration est bienvenue. En voici quelques unes :

- Vous pouvez ajouter des instructions à partir de ce langage initial. Par exemple : **CT** qui Cache la Tortue. Dans ce cas, la tortue n'est plus dessinée jusqu'à l'instruction **MT** (Montre Tortue). Ou encore **IC** qui Inverse la Couleur du crayon. S'il était blanc, il devient noir et réciproquement. Vous pouvez également laisser libre cours à votre imagination pour en trouver d'autres.

- Vous pouvez laisser à l'utilisateur le choix de la taille de la fenêtre dans laquelle évoluera la tortue. Vous ferez alors attention à ne saisir une fenêtre ni trop petite, ni trop grande.

- Vous pouvez laisser à l'utilisateur le choix entre plusieurs formes de tortue : une triangulaire et d'autres de votre choix. Mais attention, vos nouvelles tortues devront satisfaire les contraintes suivantes :

  – on doit savoir où se trouve la tête de la tortue pour savoir dans quelle direction elle peut avancer ;

  – on doit également savoir où se trouve le crayon pour pouvoir relier des traits.

  Un exemple de nouvelle tortue est représentée dans les figures 5 et 6. Cette tortue est formée de 3 cercles : un pour le corps, un pour la tête et un pour le crayon.

- Vous pouvez ajouter la possibilité de répéter un certain nombre de fois une ou plusieurs commandes. Cela est très pratique si vous voulez dessiner un carré, un hexagone, voire un cercle ! Pour un carré, l'instruction serait :
  RP 4 AV 20 TD 90
  Pour un cercle, il suffirait d'écrire :
  RP 360 AV 1 TD 1

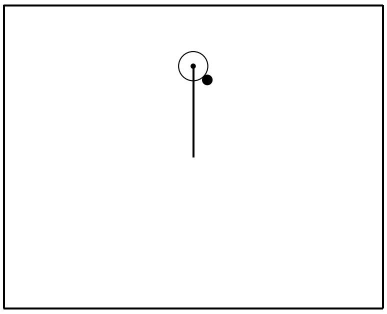


Fig 5

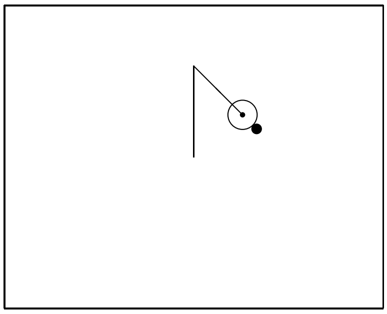


Fig 6

Cette dernière amélioration est de loin la plus utile pour pouvoir réaliser des dessins un peu compliqués, mais vous êtes libres de préférer d'autres améliorations à celles-ci.

Implémentation

Vous avez vu que toutes les commandes contiennent exactement 2 lettres. C'est pour vous faciliter la saisie. Et si vous êtes certains dès le début,

que vous ne ferez pas l'amélioration **répète**, vous pouvez même imposer que chaque ligne ne contienne qu'une seule commande. Par contre, si vous voulez vous garder la possibilité de réaliser simplement l'amélioration **répète**, vous avez intérêt à savoir traiter plusieurs ordres écrits sur une même ligne. Par exemple :
TD 90 AV 30 LC AV 10 BC AV 10
En effet, l'instruction répète fonctionnera de la manière suivante :
RP N I1 I2 I3 . . .
consistera à répéter N fois la séquence I1, I2, I3, . . . (où I1, I2, I3, . . . sont des ordres connus de la tortue). Il faudra donc

- lire et exécuter I1, lire et exécuter I2, . . .
- recommencer à lire et exécuter I1, . . .
- . . .

Et cela est à faire N fois.

Pour cela, on lira la ligne entière pour la stocker dans un tableau de caractères. Une variable entière *curseur* nous dira constamment où l'on se trouve sur le tableau de caractères. Par exemple, si on vient de traiter **LC** dans l'exemple ci-dessus, on aura : $curseur = 14$.

Après avoir traité un ordre, on incrémentera simplement *curseur* pour traiter l'ordre suivant.

- Si la ligne ne commence pas par un **RP N** alors, quand le curseur sera au bout de la chaine de caractères, on lira la ligne suivante pour continuer.
- Si la ligne commence par exemple par **RP 4**, il suffira de parcourir la ligne 4 fois avant de passer à la ligne suivante. Après chaque parcours, le curseur sera placé juste après le **RP 4** : ($curseur = 4$;).

Pour vous simplifier la tâche, vous pouvez imposer une séparation de chaque ordre par un et un seul espace.
Dans tous les cas, vos choix devront apparaitre clairement dans le document joint au projet.

ISOLA

Sujet proposé par Pierre Vinant

Il s'agit d'un jeu à deux joueurs.

Description du jeu

Le matériel

- Un plateau carré de 36 cases initialement blanches (6 x 6).
- Deux pions différents : un par joueur.
- Des carrés noirs pour interdire certaines cases.

Les règles du jeu

La position initiale des pions est laissée au choix des joueurs. Les adversaires jouent à tour de rôle. Jouer consiste à effectuer les deux étapes suivantes :

- Déplacer son pion sur l'une des cases blanches voisines qui n'est pas occupée par le pion adverse. Il n'est donc pas possible de se déplacer sur une case noire, ni sur la case occupée par le pion adverse. Un pion a donc au maximum 8 possibilités de déplacement.
- Interdire l'accès d'une case en y plaçant un carré noir. Evidemment, il n'est possible de placer un carré noir ni sur une case occupée par un pion, ni sur une case noire.

Voici un tout début de partie. La figure 1 représente la position initiale choisie par les deux joueurs. Les deux joueurs sont O et S. C'est O qui commence à jouer dans la figure 2. X symbolise la case interdite par O.

	1	2	3	4	5	6
A						
B		O				
C						
D					S	
E						
F						

Fig 1

	1	2	3	4	5	6
A						
B						
C			O		X	
D					S	
E						
F						

Fig 2

Après ce remarquable coup, S ne peut plus se déplacer en C-5 ...

But du jeu

Un joueur a perdu, si, quand vient son tour, il ne peut pas se déplacer. C'est à dire quand toutes les cases voisines de son pion sont soit noires, soit occupées par le pion adverse. Voici deux exemples de fin de partie :

	1	2	3	4	5	6
A						
B				X		
C			X		X	X
D		X		X	X	O
E		X	S		X	X
F			X			

Fig 3

	1	2	3	4	5	6
A						
B		X	X	X		X
C		X	S	X		
D		X	X	O		X
E			X		X	
F				X		

Fig 4

Dans la figure 3, O ne peut plus jouer, donc S a gagné. Dans la figure 4, O vient de jouer et gagne car le pion O bouche la dernière possibilité de déplacement de S.

Description du projet

Le projet consiste :

1. à fournir les outils pour pouvoir jouer :
   - dessin du plateau de jeu,
   - dessin des pions que l'on peut déplacer à tour de rôle selon les règles énoncées ci-dessus,
   - possibilité pour chaque joueur, après qu'il eût déplacé son pion, de noircir une case de son choix (tout en respectant les règles),
   - le programme doit détecter la fin de la partie quand elle survient.
2. à donner la possibilité à l'utilisateur, de jouer contre l'ordinateur. Il pourra choisir de jouer avec un ami humain, ou bien avec votre programme. Il va donc falloir que vous écriviez un programme qui puisse jouer à Isola le mieux possible. Et si vous êtes plusieurs à choisir ce projet, on pourra comparer vos différentes stratégies en faisant jouer vos programmes les uns contre les autres !

Améliorations possibles

Toute amélioration est la bienvenue. En voici trois possibles :

- Vous pouvez laisser le choix de la taille du plateau de jeu à l'utilisateur. Il peut en effet être intéressant d'essayer de jouer sur des plateaux de taille différente (et même des plateaux rectangulaires : 8x6 ...).

- Vous pouvez utiliser une fonction aléatoire pour éviter que le programme ne choisisse toujours la même position initiale. Cela peut aussi servir à choisir un déplacement quand plusieurs possibilités sont considérées comme équivalentes par le programme.

- Il existe une autre version de ce jeu légèrement différente où seuls les déplacements des pions sont modifiés. Ils ne se déplacent plus sur une case adjacente, mais à la manière d'un cavalier du jeu d'échec. Vous pouvez donc laisser le choix du mode de déplacement à l'utilisateur : type classique ou type cavalier. On peut même imaginer que les deux joueurs puissent choisir des modes différents !

Si vous avez d'autres idées d'amélioration, vous pouvez bien sûr les réaliser sans faire celles proposées.

Implémentation

Pour saisir les déplacements des pions et les cases que l'on noircit, vous pourrez numéroter les colonnes et mettre des lettres sur les lignes. Ainsi, chaque case aura un nom : (C-5), (E-3), ... Il faudra alors faire une transcription pour transformer le nom de la case en un couple de coordonnées dans un tableau C.

Le plateau pourra être représenté dans votre programmme par un tableau de caractères. Une case sera *'P'* si elle est occupée par un pion, *'N'* si c'est une case noire, ou *'B'* dans le cas d'une case blanche. Voici donc une déclaration du type plateau :

```
typedef char[6][6] Plateau;
```

Vous aurez besoin de la position des pions et peut-être du dessin de chaque pion. Voici une déclaration possible :

```
typedef struct pion {
                      int ligne;
                      int colonne;
                      char  dessin ;
                    } Pion;
```

Si pion1 est une variable de type Pion et si le dessin du pion1 est un rond, l'on aura par exemple : `pion1.dessin = 'R'` (R pour Rond). Mais vous n'êtes bien sûr pas obligés de choisir cette structure de données.

Comment faire jouer l'ordinateur ?

Pour choisir le meilleur déplacement du pion, vous pouvez noter les différentes positions possibles.
Exemple de notation :

- une case occupée ou noire aura une note négative (-1) ;
- les autres auront comme note le nombre de cases blanches vides adjacentes à la case à noter.

Le choix du programme ira vers la case ayant la plus grande note. Mais cette notation n'est pas forcément la meilleure. Peut-être trouverez-vous mieux ?

Pour interdire une case, vous pourrez utiliser le même principe. Une fonction évaluera la case la plus gênante pour l'adversaire et si possible la moins gênante pour soi. A vous de trouver une fonction approchant ces objectifs.

Calcul du polynôme caractéristique d'une matrice par la méthode de Faddeev

Sujet proposé par Jean-Yves Thibon

Le polynôme caractéristique

Soit A une matrice carrée $n \times n$ à coefficients réels ou complexes. On appelle **polynôme caractéristique** de A le polynôme $P_A(x)$ défini par le déterminant suivant

$$P_A(x) = \begin{vmatrix} a_{11} - x & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} - x & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} - x \end{vmatrix} = det(A - xI)$$

où I est la matrice identité d'ordre n. Par exemple,

$$A = \begin{pmatrix} 3 & 4 \\ 5 & 2 \end{pmatrix} \quad \longrightarrow \quad P_A(x) = \begin{vmatrix} 3-x & 4 \\ 5 & 2-x \end{vmatrix} = x^2 - 5x + -14.$$

L'objectif du projet est de programmer une méthode de calcul du polynôme caractéristique qui ne nécessite pas de développement de déterminant. Cette méthode, due au mathématicien russe D.K. Faddeev, n'utilise que les opérations les plus simples sur les matrices (somme, produit, multiplication par une constante, trace). C'est une modification astucieuse d'une méthode due à l'astronome Leverrier, qui l'avait utilisée pour les calculs qui l'ont conduit à la découverte de la planète Neptune.

Fonctions symétriques et méthode de Leverrier

Ce paragraphe explique le principe de la méthode, il n'est pas indispensable de l'étudier immédiatement. L'algorithme à implémenter est décrit au paragraphe suivant. On démontre que si $\lambda_1, \lambda_2, \dots, \lambda_n$ sont les racines (éventuellement complexes) de $P_A(x)$, il existe une matrice inversible P telle que $T = P^{-1}AP$ soit une matrice triangulaire de la forme

$$T = \begin{pmatrix} \lambda_1 & \star & \dots & \star & \star \\ 0 & \lambda_2 & \ddots & & \star \\ \vdots & \ddots & \ddots & & \vdots \\ 0 & \dots & \dots & 0 & \lambda_n \end{pmatrix}$$

avec les λ_i sur la diagonale. On rappelle que la **trace** d'une matrice est la somme $\mathrm{trA} = \sum_{i=1}^{n} a_{ii}$ de ses coefficients diagonaux, et qu'on a toujours $\mathrm{trAB} = \mathrm{trBA}$, ce qui entraîne $\mathrm{trP^{-1}AP} = \mathrm{trA}$. On a donc

$$\mathrm{trA} = \mathrm{trT} = \lambda_1 + \lambda_2 + \dots + \lambda_n$$

et plus généralement

$$\mathrm{trA^k} = \mathrm{trT^k} = \sum_{\mathrm{i=1}}^{\mathrm{n}} \lambda_{\mathrm{i}}^{\mathrm{k}}$$

car on vérifie facilement que T^k est encore triangulaire, avec les λ_i^k sur la diagonale.

Posons

$$P_A(x) = (-1)^n \prod_{i=1}^{n} (x - \lambda_i) = (-1)^n (x^n - \sigma_1 x^{n-1} + \sigma_2 x^{n-2} - \cdots + (-1)^n \sigma_n)$$

les σ_i étant les **fonctions symétriques élémentaires** des racines λ_i, c'est-à-dire

$$\sigma_1 = \sum_i \lambda_i, \qquad \sigma_2 = \sum_{i<j} \lambda_i \lambda_j, \ldots$$

$$\sigma_k = \sum_{i_1<i_2 \ldots <i_k} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k}, \ldots, \qquad \sigma_n = \lambda_1 \lambda_2 \cdots \lambda_n$$

(**formules de Viète**).

Les σ_i, et donc les coefficients du polynôme $P_A(x)$, peuvent s'exprimer au moyen des **sommes de puissances** s_k, définies par

$$s_k = \sum_{i=1}^{n} \lambda_i^k = \mathrm{trA^k},$$

la relation étant donnée par les **formules de Newton** :

$$k\sigma_k = s_1 \sigma_{k-1} - s_2 \sigma_{k-2} + \cdots + (-1)^{i-1} s_i \sigma_{k-i} + \cdots + (-1)^k s_k,$$

avec $\sigma_0 = 1$ (par définition).

Par exemple, $\sigma_1 = s_1, \sigma_2 = \frac{1}{2}(s_1^2 - s_2)$, et de la relation $3\sigma_3 = s_1\sigma_2 - s_2\sigma_1 + s_3$ on tire $\sigma_3 = \frac{1}{6}s_1^3 - \frac{1}{2}s_1 s_2 + \frac{1}{3}s_3$.

La méthode de Leverrier consistait à calculer les coefficients de $P_A(x)$ en partant des $s_k = \mathrm{trA^k}$ et en utilisant les formules de Newton.

L'algorithme de Faddeev

Il fait appel au même principe que la méthode de Leverrier, mais les calculs sont organisés de manière plus astucieuse. On construit par récurrence les suites (de matrices et de nombres) :

$$\begin{array}{ccc}
A_1 = A & p_1 = \mathrm{trA_1} & B_1 = A_1 - p_1 I \\
A_2 = B_1 A & p_2 = \frac{1}{2}\mathrm{trA_2} & B_2 = A_2 - p_2 I \\
\ldots & \ldots & \ldots \\
A_k = B_{k-1} A & p_k = \frac{1}{k}\mathrm{trA_k} & B_k = A_k - p_k I \\
\ldots & \ldots & \ldots \\
A_n = B_{n-1} A & p_n = \frac{1}{n}\mathrm{trA_n} & B_n = A_n - p_n I
\end{array}$$

On peut alors démontrer que le polynôme cherché $P_A(x)$ est donné par

$$P_A(x) = (-1)^n(x^n - p_1x^{n-1} - p_2x^{n-2} - \cdots - p_n) = (-1)^n(x^n - \sum_{i=1}^{n} p_i x^{n-i}).$$

De plus, si la matrice A est inversible, son inverse est donné par

$$A^{-1} = \frac{1}{p_n}B_{n-1}.$$

Le projet

On demande d'implémenter l'algorithme de Faddeev. On définira un type `matrice` permettant de manipuler des matrices carrées, et on écrira des fonctions `mult`, `add`, `soust` permettant de multiplier, additionner et soustraire deux matrices carrées, une fonction `trace` calculant la trace, et une fonction `subxid` permettant de soustraire directement un multiple de la matrice I à une matrice donnée.

Le programme devra être aussi modulaire que possible. En particulier, le polynôme caractéristique devra être calculé par une fonction qui renvoie le tableau des coefficients p_i. La lecture des données et l'impression du résultat devront faire l'objet de fonctions séparées. On pourra prévoir une option qui affiche l'inverse de la matrice donnée lorsqu'elle est inversible.

Un test intéressant pour l'ensemble des fonctions est fourni par le **théorème de Cayley-Hamilton**, qui affime que lorsque l'on substitue à x dans $P_A(x)$ la matrice A elle-même (le terme constant $-p_n$ étant interprété comme la matrice diagonale $-p_nI$), on obtient la matrice nulle. On écrira une fonction `evalpolmat` permettant d'évaluer un polynôme quelconque en une matrice donnée, et on l'utilisera pour calculer $P_A(A)$.

Manipulation de polynômes

sujet proposé par Marc Zipstein

Le but du projet est d'implémenter des fonctions de manipulation de polynômes à coefficients dans $\mathbb{Q}$. Un polynôme sera représenté comme un tableau de monômes. Le type de base `monom` est :

```
typedef mon {
            int degre,num,den;
          } Monome;
```

(degré du monôme, numérateur et dénominateur du coefficient).
Un polynôme est une structure composée d'un tableau de monômes et d'un entier (on ne mémorise pas les termes de coefficients nuls) :
`typedef struct poly {`
`Monome monome[MAX];` (`MAX` constante prédéfinie)
`int nbmo;` (nombre de monômes non nuls formant le polynôme)
`} Polynome;`

Exemple : le polynôme $P = -\frac{1}{7} + 2x^2 + \frac{3}{4}x^5 - x^{12}$ est représenté par :

Polynôme

degre	num	den
0	-1	7
2	2	1
5	3	4
12	-1	1
?	?	?
?	?	?

`nbmo` $= 4$

Le degré des polynômes traités n'est donc pas limité par la taille du tableau. En revanche, le nombre de monômes composant un polynôme est limité (par `MAX`+1).

Les différentes fonctions à implémenter sur les polynômes seront :

- Lecture ;
- Affichage ;
- Affectation ;
- Evaluation en un point ;
- Dérivée ;
- Primitive s'annulant en un point (entré au clavier) ;

- Somme, différence ;
- Produit, carré ;
- Division euclidienne ;
- Détermination des polynômes coefficients dans l'égalité de Bezout pour deux polynômes (cf annexe p. 23).

Ces différentes fonctions seront accessibles à partir d'un menu. II sera particulièrement tenu compte de la facilité d'utilisation du programme et de la présentation des résultats. On devra en particulier pouvoir définir et utiliser 10 polynômes reconnus par leurs noms : $A, B, C, \ldots, J$. (Exemple : affecter à J la somme de A et B).

Cryptage du sac à dos

sujet proposé par Marc Zipstein

Problème

Etant donné un ensemble $E = \{x_1, x_2, \ldots, x_n\}$ d'entiers strictement positifs et un entier strictement positif S, le problème du sac à dos consiste à trouver, s'il existe, un sous-ensemble P de E tel que la somme des éléments de P soit S.

On ne connaît pas, dans le cas général, d'autres solutions à ce problème que d'essayer tous les sous-ensembles de E (il y en a 2^n). Cependant, il existe un cas particulier où ce problème est facile à résoudre. Celui où les x_i forment une suite super-croissante. Une suite (x_n) est dite **super-croissante** si et seulement si chaque élément est supérieur à la somme de tous les précédents. Dans ce cas, le problème du sac à dos peut être traité par :

tant que $x_1 \leq S$

  sélectionner le plus grand élḿent x_j tel que $x_j \leq S$;

  remplacer S par $S - x$, P par $P \bigcup \{x_j\}$, E par $E - \{x_j\}$

si $S = 0$ le problème est résolu;

sinon le problème n'a pas de solution.

Ce problème est à la base d'une méthode de cryptage dite à clef publique, permettant de crypter des suites de n bits. Un **cryptage à clef publique** est un cryptage dans lequel chaque personne A met à la disposition du public une clef C_A. Pour envoyer un message à A on le crypte à l'aide de cette clef C_A. Si la méthode de cryptage est bonne, seul A est capable de décrypter le message.

Principe

La personne A détermine une suite super-croissante $s_1, \ldots, s_n$. Elle choisit un nombre premier T plus grand que la somme des s_i et une valeur V telle que $T/2 < V < T$; il existe donc W tel que $VW = 1 \pmod{T}, W$ est l'inverse de V dans $\mathbb{Z}/T\mathbb{Z}$. (W peut être déterminé grâce à l'identité de Bezout, cf. annexe).

A détermine la suite $a_i = s_i \times V \pmod{T}$. La suite $(a_1, \ldots, a_n, T)$ constitue la clef publique publiée par A (remarque : la suite (a_i) n'est plus super-croissante).

Pour envoyer un message $m = (b_1, \ldots, b_n)$ de n bits à A, on calcule l'entier

$$S = b_1 \times a_1 + b_2 \times a_2 + \ldots + b_n \times a_n.$$

Le décryptage pour une personne autre que A est un problème difficile (2^n essais). Par contre, lorsque A reçoit S il calcule $S^{'} = S \times W \pmod{T}$. Or,

$$\begin{aligned}
S^{'} &= b_1 \times a_1 \times W + b_2 \times a_2 \times W + \ldots + b_n \times a_n \times W \pmod{T} \\
&= b_1 \times s_1 \times W \times V + b_2 \times s_2 \times W \times V + \ldots + b_n \times s_n \times W \times V \pmod{T} \\
&= b_1 \times s_1 + b_2 \times s_2 + \ldots + b_n \times s_n \pmod{T}.
\end{aligned}$$

Comme la suite des s_n est super-croissante, il est facile de déterminer les s_i intervenant dans la somme et donc les valeurs des b_i.

Exemple : A choisit $E = \{3, 7, 11, 22, 45, 103, 213, 417\}, T = 1013$ et $V = 538$, il calcule $W = 804$ et publie $(601, 727, 853, 693, 911, 712, 125, 473)$.

Pour envoyer le message (01100001) à A, on calcule :

$$\begin{aligned}
&0 \times 601 + 1 \times 727 + 1 \times 853 + 0 \times 693 \\
&+0 \times 911 + 0 \times 712 + 0 \times 125 + 1 \times 473 = 727 + 853 + 473 \\
&= 2053.
\end{aligned}$$

"2053" est le message envoyé à A. Lorsque A reçoit le message, il calcule $2053 \times 804 \pmod{1013} = 435$.

$435 = 0 \times 3 + 1 \times 7 + 1 \times 11 + 0 \times 22 + 0 \times 45 + 0 \times 103 + 0 \times 213 + 1 \times 417$

A retrouve le message (01100001).

Le projet

Un contrôle de dépassement de `LONG_MAX = 2147483647` (plus grand entier représentable en C sur Merlin devra être effectué lors des calculs.

On demande de construire un programme de cryptage pour $n = 8$. Il devra :

- aider l'utilisateur à construire une suite super-croissante ;
- proposer des nombres premiers T ;
- lire V, déterminer son inverse W, ainsi que la clef publique (a_i) ;
- permettre de crypter un groupe de 8 bits en un entier ;
- permettre de décrypter un entier en un groupe de 8 bits.

Les caractères sont codés sur la machine par groupes de 8 bits. La valeur associée à une lettre est son code ASCII. Ce code s'obtient directement en C en considérant le caractere comme un entier :
`printf("le caractere %c a pour code %d \n",'A','A')}` affiche :

```
le caractere A a pour code 65
```

L'écriture de 65 en binaire est (01000001). De même le code ASCII de 'a' est 97 qui s'écrit (01100001) en binaire. Adapter le programme pour ne plus coder des suites de bits mais directement des lettres.

Remarque : Coder des suites de 8 bits n'est pas un cryptage sûr (il suffit de 256 essais pour décrypter). Cependant, même avec de suites de 100 bits, il existe des méthodes qui permettent de casser ce cryptage sans calculer toutes les possibilités !

Options :

- Coder les lettres deux par deux (ou plus) et donc les bits par groupe de 16 (ou plus).
- Ecrire une fonction permettant de décrypter un message codé par essais successifs.

Annexe : détermination des entiers intervenant dans l'identité de Bezout

Egalité de Bezout : Soient deux entiers a et b et soit d leur pgcd. Il existe un couple (u, v) d'entiers relatifs tels que :

$$au + bv = d\ .$$

Méthode de détermination des entiers u et v de l'égalité de Bezout : Le pgcd étant le dernier reste non nul dans l'algorithme d'Euclide, la méthode consiste à écrire les restes successifs en fonction de a et de b. On ne fait qu'appliquer $a = b \times q + r \Leftrightarrow r = a - b \times q$ et à reporter tout au long des calculs !

Notations : On définit une suite $a_0 = a, a_1 = b, a_{i+2} = a_i \pmod{a_{i+1}}, a_k$ est le dernier reste non nul (pgcd) et deux suites (u_i) et (v_i) telles que pour tout i on ait :

$$a \times u_i + b \times v_i = a_i$$

L'idée est de maintenir tout au long du calcul des valeurs vérifiant la relation.

$$i = 0, a \times u_0 + b \times v_0 = a_0 \times a_1 + b_0 = a = a_0$$

$$i = 1, a \times u_1 + b \times v_1 = a_1 \times a_0 + b_1 = b = a_1$$

$$\dots$$

$$i, a \times u_i + b \times v_i = a_i$$

$$i + 1, a \times u_{i+1} + b \times v_{i+1} = a_{i+1}$$

$$i+2, a_{i+2} = a_i \pmod{a_{i+1}}$$

soit $a_i = a_{i+1} \times q + a_{i+2}$, d'où

$$\begin{array}{rcl} a_{i+2} & = & a_i - a_{i+1} \times q \\ & = & a \times u_i + b \times v_i - q(a \times u_{i+1} + b \times v_{i+1}) \\ & = & a(u_i - q \times u_{i+1}) + b(v_i - q \times v_{i+1}) \end{array}$$

Les suites (u_i) et (v_i) sont définies par

$$u_0 = 1, u_1 = 0, u_{i+2} = u_i - q \times u_{i+1}$$

$$v_0 = 1, v_1 = 0, v_{i+2} = v_i - q \times v_{i+1}.$$

où q est le quotient de la division euclidienne dans le calcul du pgcd.

Gestion de droits d'accès

sujet proposé par Marc Zipstein

On veut simuler la gestion des fichiers dans un système multi-utilisateurs, ayant les propriétés suivantes :

1. Les utilisateurs sont identifiés par des numéros ; il y a au plus `NBUTIL` = 5 utilisateurs.
2. Les fichiers sont identifiés par des numéros ; il y a au plus `NBFIC` = 100 fichiers.
3. Tout utilisateur est propriétaire des fichiers qu'il crée.
4. Un fichier peut être utilisé de trois façons différentes :
   - en lecture (ex : lecture d'un fichier source)
   - en écriture (je modifie un fichier source)
   - en exécution (je demande l'exécution d'un fichier exécutable).

   Pour accéder à un fichier dans un de ces modes, un utilisateur doit en avoir la permission (en lecture, en écriture ou en exécution).
5. Lors de la création d'un fichier le propriétaire du fichier fixe les permissions qu'il s'accorde et les permissions qu'il accorde aux autres utilisateurs.
6. A chaque instant le propriétaire d'un fichier peut modifier ces permissions.

But du projet

On réalisera un système de gestion des droits d'accès aux fichiers sans créer véritablement les fichiers. Un fichier sera représenté par une structure contenant le numéro du propriétaire du fichier, les droits d'accès du propriétaire et les droits d'accès des autres utilisateurs. Ces structures seront rangées dans un tableau de taille `NBFIC`, l'indice, dans le tableau, de la structure représentant un fichier est le numéro du fichier. Le programme possèdera les fonctionnalités suivantes :

- Login : l'utilisateur rentre son numéro et devient l'utilisateur courant.
- Logout : il n'y a plus d'utilisateur courant, le programme reste bloqué sur la demande de Login.
- Création d'un fichier, avec attribution des permissions pour le propriétaire et pour les autres.

- Modification des permissions associées à un fichier par son propriétaire.
- Demande de lecture, d'écriture, d'exécution d'un fichier. La possibilité pour un fichier donné de connaître son propriétaire et les permissions associées au fichier pour le propriétaire et pour les autres.
- La possibilité pour un utilisateur de connaître tous les fichiers dont il est propriétaire avec les permissions qui leurs sont associées.
- L'affichage de tous les numéros de fichiers avec en regard leur propriétaire et les droits qui leur sont associés.

Améliorations possibles

- Désigner les utilisateurs par leur nom ;
- Installer un système de mots de passe ;
- Installer des groupes d'utilisateurs.

Le système UNIX est plein de bonnes idées.

Fermeture transitive d'un graphe

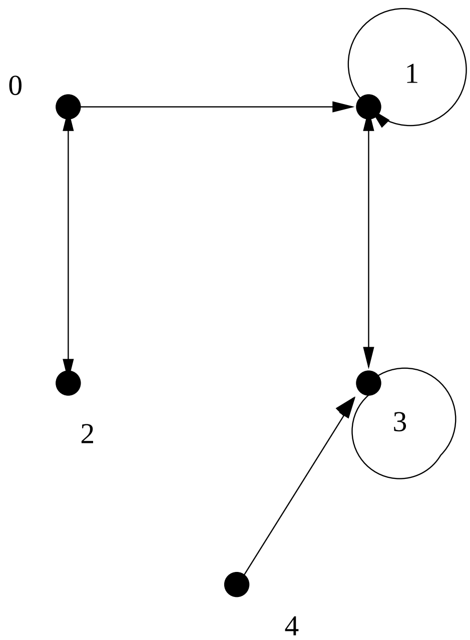
sujet proposé par Marc Zipstein

Définitions

Un **graphe** G est un couple (X, A) où :

- X est un ensemble fini de sommets ;
- A est un sous-ensemble de $X \times X$, l'ensemble des arcs de G.

On prendra généralement pour X un segment $[0, N-1]$.
Exemple :
$G = (\{0,1,2,3,4\}, \{(0,1),(0,2),(1,1),(1,3),(2,0),(3,1),(3,3),(4,3)\})$ peut être représenté graphiquement par :



Un **chemin** dans un graphe est une séquence d'arcs $(x_0, x_1)(x_1, x_2)\ldots(x_{n-1}, x_n)$ du graphe ; x_0 et x_n en sont respectivement les **extrémités initiale et terminale**. Le nombre d'arcs de ce chemin est sa **longueur**.

Dans l'exemple précédent la séquence : $(0,2)(2,0)(0,1)(1,1)(1,3)(3,3)(3,1)$ est un chemin de 0 à 1 de longueur 7.

On associe à un graphe $G = ([0, N-1], A)$ une matrice carrée d'ordre N à valeurs dans $\{0, 1\}$ telle que

$$\forall i, j \in [0, N-1], \quad m_{i,j} = 1 \Leftrightarrow (i, j) \in A.$$

La matrice associée à l'exemple précédent est :

$$\begin{array}{cccccc} & 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 1 & 0 \end{array}$$

La **fermeture transitive** d'un graphe G est le graphe G' où deux sommets i et j sont reliés par un arc si et seulement si il existe un chemin dans G allant de i à j.

La fermeture transitive du graphe précédent est donc le graphe :

$$\begin{array}{cccccc} & 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 1 & 0 \end{array}$$

Algorithme

Les méthodes de test d'existence d'un chemin entre deux sommets d'un graphe sont fondées sur le **lemme de König** qui assure que s'il existe un chemin entre deux sommets d'un graphe, il en existe alors un de longueur inférieure ou égale à n, le nombre de sommets du graphe.

Pour calculer la fermeture transitive d'un graphe, nous allons définir tout d'abord deux opérations sur les matrices à valeurs booléennes :
Soient deux matrices carrées $A = (a_{i,j})$ et $B = (b_{i,j})$ de dimension n à valeurs $\{0, 1\}$, on définit leur somme S et leur produit P par les formules suivantes :

$$s_{i,j} = \min(1, a_{i,j} + b_{i,j})$$

$$p_{i,j} = \min(1, \sum_{1 \leq k \leq n} (a_{i,k} * b_{k,j}))$$

Pour calculer la fermeture transitive d'un graphe G de N sommets associé à une matrice M, on définit une suite de matrices d'ordre n, à valeurs dans $\{0, 1\}$ par :

$$M_1 = M, \quad M_p = M * M_{p-1} \ .$$

On montre qu'il existe un chemin de longueur k entre deux sommets i et j de G si et seulement si on a $M_k(i, j) = 1$. Cette remarque combinée avec le lemme de König fournit un algorithme : la somme booléenne des n matrices $M_1, M_2, \ldots, M_n$ n'est autre que la matrice associée à la fermeture transitive de G.

Problème

On écrira un programme qui possède les fonctionnalités suivantes :

- saisie de la matrice associée à un graphe G ;
- affichage de la matrice de la fermeture transitive de G ;
- réponse à la question : existe-t-il un chemin de longueur k entre les sommets i et j ?

Variations sur la commande `wc`

Sujet proposé par Ahmad Daaboul

La commande `wc` sous Unix permet de compter le nombre de caractères, de mots et de lignes dans un texte donné.

Problème

On ne mémorise qu'un seul texte dans un tableau de `MAX` caractères (`MAX` étant une constante définie, 10000 caractères par exemple).

Ecrire un programme permettant à un utilisateur d'entrer des textes puis d'effectuer sur les textes entrés, les opérations suivantes selon un système de menu :

- Créer un nouveau texte (`C`) ;
- Afficher le texte courant (`a`) ;
- Calculer et afficher le nombre de mots (`m`) ;
- Calculer et afficher le nombre de caractères (`c`) ;
- Calculer et afficher le nombre de lignes (`l`) ;
- Calculer et afficher la fréquence d'une lettre donnée (`f lettre`) ; (quelle que soit la lettre, ne pas afficher de fréquence nulle) ;
- Calculer et afficher la fréquence d'un mot (F mot)
- Question Oulipo (`o`).

Le **jeu Oulipo** consiste à écrire une procédure permettant de construire un texte (T2) à partir d'un premier texte (T1) (le texte courant). La procédure reçoit en paramètre les entiers suivants :

- *pos*, la position dans T1 du premier mot qui sera pris comme référence,
- *taille*, le nombre de mots voulu dans T2.

Les mots de T2 sont construits de la façon suivante : Le premier mot de T2 correspond au *pos*-ième mot de T1. Soit M ce mot. Ensuite, on recherche dans T1, la première nouvelle occurrence de M. Si elle existe, M devient le mot qui suit cette nouvelle occurrence; sinon, M devient le mot qui le suit. Dans les deux cas, on écrit dans T2 la nouvelle valeur de M, et on recommence le processus avec cette nouvelle valeur de M. Si on arrive à la fin de T1, on repart au début jusqu'à la taille donnée.

Exemple

<u>T1 :</u>
deux et deux quatre
quatre et quatre huit
huit et huit font seize
répétez dit le maître

m : 17
c : 71
l : 4
f z : 2
F deux : 2

Oulipo
pos : 2
taille : 5
M est : et
La nouvelle valeur de M est : quatre
T2 : et quatre quatre huit et deux.

Puissance 4

Sujet proposé par Daniel Hirschkoff

Puissance 4 est un jeu à deux joueurs composé d'une grille verticale de 10 cases sur 10 (par exemple), dans laquelle les joueurs font tomber chacun à leur tour un pion de leur couleur dans une colonne de la grille, remplissant ainsi la case libre la plus basse de la colonne correspondante. Le gagnant est le premier joueur qui arrive à aligner 4 pions de sa couleur horizontalement, verticalement, ou en diagonale. À puissance 4, pour une raison étrange, les joueurs ne sont pas noir et blanc, mais jaune et rouge. Le programme devra afficher la grille de jeu, et ensuite proposer un menu comportant trois choix :

- coup du joueur jaune (si c'est au jaune de jouer, sinon afficher “coup du joueur rouge”, bien entendu) ;
- un coup en arrière ;
- quitter.

On entre un coup en indiquant la colonne dans laquelle le jouer fait tomber son pion. Si on choisit l'option “coup en arrière”, on n'a plus accès lors de l'entrée suivante à cette option (autrement dit, on ne peut faire qu'un seul coup en arrière, on n'a pas accès à tout l'historique de la partie). L'option “quitter” sert à interrompre une partie où il est évident qu'aucun des deux joueurs ne pourra gagner.

Lorsqu'un coup est rentré, le système calcule si c'est un coup gagnant (autrement dit si une ligne de 4 pions a été créée), et dans le cas contraire affiche à nouveau le menu ci-dessus.

L'affichage de la grille se fera en utilisant les fonctions de la bibliothèque graphique.

Améliorations possibles

- Paramètrer le programme de manière à ce que l'on puisse choisir de jouer à puissance n avec n choisi par l'utilisateur : vous verrez rapidement que le choix le plus malin est tout de même 4, car les parties de puissances 2 ne durent jamais longtemps, et les parties de puissance 10 sont souvent nulles.
- Gérer l'historique des coups joués, de manière à pouvoir retourner plusieurs coups en arrière dans une partie.
- Faire jouer l'utilisateur contre le programme. On pourra s'inspirer des méthodes proposées dans d'autres projets.

Répertoire Téléphonique

Sujet proposé par Bruno Gauthier

Objectif du Projet

On se propose de gérer l'utilisation du répertoire d'un réseau téléphonique. Pour plus d'informations, consulter le serveur **Web** de la société **France Télécom**[1] ! Le but sera de proposer un programme permettant de traiter de nombreuses requêtes (ajout/retrait d'un abonné, recherche d'un numéro, etc...).

Description

Nous définissons tout d'abord les propriétés du répertoire :

- les numéros d'appels du répertoire sont codés sur 2 chiffres (de 00 à 99) ;
- un numéro est affecté ou non. S'il est affecté, il l'est à un correspondant dont le nom est précisé ;
- la longueur des noms des correspondants sera limitée à 20 caractères.

Typage des données

Voici quelques définitions de type qui permettent de coder les éléments de base du répertoire :

```
#define NoAppelMAX 99+1  /* pourquoi +1 ? */
#define NomLongMAX 20+1  /* pourquoi +1 ? */

typedef struct
      {
        int  Affect;
        char Nom[NomLongMAX];
      } NoAbonne;


NoAbonne Repert[NoAppelMAX];
```

[1] http://www.francetelecom.fr

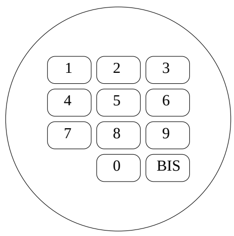
Principe

Donner un programme qui réalise les commandes suivantes :

- affectation, réaffectation ou libération d'un numéro d'appel (modification interne du répertoire) ;
- visualisation de la liste des numéros libres, ou celle des numéros affectés avec les nom des correspondants ;
- visualisation des noms des correspondants par ordre alphabétique ;
- recherche du nom d'un correspondant dont le numéro est `N` ;
- recherche du numéro d'appel d'un correspondant `X` ;
- recherche du numéro d'appel d'un correspondant dont une partie du nom est `X'` (expliquer précisément votre algorithme de recherche).

Travail Demandé

Le programme devra pouvoir répondre aux questions citées dans la section précédente, notamment à l'aide d'une interface graphique qui sera composée notamment d'un clavier téléphonique du type suivant :



Le rapport de projet devra comprendre :

- l'analyse du problème ;
- la description des algorithmes utilisés ;
- le programme et ses commentaires ;
- aide et documentation pour l'utilisateur.

La modularité du programme fera l'objet d'une attention particulière lors de la réalisation du projet.

Options proposées :

- gestion d'une liste rouge ;
- gestion de plusieurs lignes par abonnés (téléphone/fax/bibop/portable/etc...) ;
- toute autre amélioration est la bienvenue !

Master Mind Graphique

Sujet proposé par Pierre Vinant

Le but du projet est de réaliser un Master Mind graphique. L'utilisateur cherchera la combinaison cachée par le programme. A chaque étape, le programme notera la combinaison proposée, d'une des trois manières suivantes (au choix par l'utilisateur).

- notation pour débutants : A chaque pion de la combinaison, est associée une case de notation. Si ce pion est bien placé, on y mettra un noir. Si ce pion est dans la combinaison cachée mais pas à cette place, on y mettra un blanc. Et dans les autres cas, cette case restera vide.
- notation classique : Aucune case de notation n'est liée à un pion particulier. Dans la notation, le nombre de noirs correspond au nombre de pions bien placés de la combinaison. Et le nombre de blancs correspond au nombre de pions figurant dans la combinaison mais à une mauvaise place.
- notation difficile : Cette notation n'utilisera qu'une seule couleur (blanc par exemple). Le nombre de blancs correspondra au nombre de pions qui apparaissent dans la combinaison, qu'ils soient bien ou mal placés.

Attention, dans les trois cas, un pion de la combinaison proposée ne sera jamais noté 2 fois, même s'il y a 2 pions de sa couleur dans la combinaison cachée. On le notera par ordre de priorité bien placé ou mal placé. Et réciproquement, si dans la combinaison cachée figure un pion rouge, et dans la combinaison proposée figure deux pions rouges, alors un seul sera noté comme bon. La priorité reste la même. Vous pourrez bien sur trouver d'autres manières de noter. L'utilisateur pouvant changer son choix avant chaque partie.

Sachant que les écrans que vous utilisez sont en noir et blanc, vous remplacerez les couleurs du Master Mind par des formes, des dessins ou des caractères. Pour la notation, vous pouvez soit laisser en noir et blanc, soit trouver autre chose. Dans la suite, *couleur* représentera ce par quoi vous le remplacerez.

Le programme devra laisser les choix suivants à l'utilisateur :

- Le nombre de couleurs différentes pouvant être utilisées dans une combinaison. Vous pouvez éventuellement laisser le choix des couleurs.
- Le nombre de pions qui constituent une combinaison (3, 4, 5, ...).
- Le nombre maximum de coups laissés à l'utilisateur pour trouver la combinaison.

- Si la combinaison cachée peut contenir des doubles, des triples, ...
- Le mode de notation (à choisir parmi les trois ci-dessus, et éventuellement d'autres).

Le programme devra tenir à jour 3 tableaux de scores, un pour chaque mode de notation. Ces tableaux de scores seront évidemment sauvés dans un fichier (ou plusieurs). Vous choisirez vous-même la manière de compter les points. Pour cela, vous pourrez tenir compte, entre autre, du temps mis par le joueur pour découvrir la solution et du nombre de combinaisons proposées.

Améliorations possibles :

- Vous pouvez essayer de faire trouver par le programme la combinaison cachée par l'utilisateur. C'est une belle amélioration, mais elle est non triviale !
- Selon les options choisies par le joueur, il sera plus ou moins facile de trouver la bonne combinaison. Donc le meilleur score ne sera pas accessible avec certaines options. Pour résoudre ce problème, vous pourrez
  - soit gérer d'autres tableaux de scores selon les options choisies (nombre de couleurs, nombre de pions d'une combinaison, si des couleurs peuvent être doublés),
  - soit moduler le calcul des scores selon les options choisies.
- Toute autre amélioration est bienvenue.

Remarque : Le programme devra pouvoir afficher la combinaison cachée dès le début afin de tester les différents modes de notation.

Entiers en précision arbitraire

Frédérique Bassino

Objet : le but du projet consiste à écrire un ensemble de fonctions permettant de manipuler des entiers en précision arbitraire.

Entiers non signés

Un entier non signé en précision arbitraire sera codé en base 100. On remarquera que coder un nombre en base 100 consiste à l'écrire d'abord en base 10, puis à regrouper les chiffres par deux en commençant par la droite. L'entier sera représenté par un tableau. Chaque élément du tableau contiendra un chiffre du codage en base 100,*i.e.*, un nombre compris entre 0 et 99. Le codage commencera par la droite. Le premier élément du tableau codera le chiffre des unités en base 100, c'est à dire les chiffres des unités et des dizaines en base 10. Le second élément du tableau codera le chiffre des centaines en base 100, c'est à dire les chiffres des centaines et des milliers en base 10 et ainsi de suite.

Types de données

On utilisera les constantes et les types suivants :

```
#define BASE   100
#define DIGIT_MAX (BASE -1)

typedef    struct _entier_non_sign
                     {
                     int  chiffres[MAX_LONG];  /* digits du nombre
                     int  compteur; /*nombre de cases occupées */
                      } Entier_Non_Signe;
```

Fonctions

Chaque fonction aura un paramètre *résultat* passé par adresse. On utilisera des tests pour s'assurer de la cohérence des calculs effectués, le résultat fonctions concernées sera un entier indiquant la validité du résultat.
Par exemple, $somme(0, 0, c)$ affecte 0 à c et retourne VRAI,
alors que $somme(Maxint, 1, c)$ affecte une valeur non pertinente à c et retourne FAUX.
Vous devez écrire les fonctions suivantes :

- une fonction d'entrée au clavier d'un entier non signé arbitrairement long;

- une fonction de sortie permettant d'afficher un entier;
- une fonction de comparaison avec zéro (test d'égalité);
- une fonction de comparaison prenant en argument deux entiers et rendant une valeur booléenne.(test d'égalité);
- une fonction d'addition prenant en argument deux entiers non signés et calculant leur somme;
- une fonction de soustraction prenant en argument deux entiers et calculant leur différence. On supposera *a priori* que le premier entier est supérieur au deuxième;
- une fonction de multiplication prenant en argument un entier compris entre 0 et 99 (inclus) et un entier en précision arbitraire et calculant leur produit;
- une fonction de multiplication prenant en argument deux entiers en précision arbitraire et calculant leur produit.

Entiers signés

Un entier signé en précision arbitraire sera representé par son signe et sa valeur absolue.

Types de données

```
typedef    struct _entier_sign
                        {
                        int sign;
                        int  chiffres[MAX_LONG];  /* digits du nombre
                        int  compteur; /*nombre de cases occupées */
                         } Entier_Signe;
```

Fonctions

Vous devez écrire les fonctions suivantes :

- Une fonction d'entrée au clavier d'un entier signé arbitrairement long.
- Une fonction de sortie permettant d'afficher un entier signé.
- Une fonction de comparaison avec zéro (test d'égalité).

- Une fonction testant si un entier est positif.
- Une fonction de comparaison entre deux entiers signés rendant une valeur utilisable comme booléen.
- Une fonction d'addition prenant en argument deux entiers signés et calculant leur somme.
- Une fonction de soustraction prenant en argument deux entiers signés et calculant leur différence.
- Une fonction de multiplication prenant en argument deux entiers signés et calculant leur produit.
- Une fonction calculant $n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2$. L'argument de cette fonction sera donné sous forme d'un entier ordinaire du C (`type int`).
- Une fonction calculant $C(n,p) = \frac{n!}{p!\,(n-p)!}$. Pour éviter les divisions, on utilisera la formule $C(n,p) + C(n,p+1) = C(n+1,p+1)$. Les arguments de cette fonction seront donnés sous forme de deux entiers ordinaires du C (type `int`).

Il est également demandé d'écrire un menu pour l'utilisation des fonctions ci-dessus.

L'écriture des fonctions suivantes est facultative :

- Calcul du quotient de la division euclidienne.
- Calcul du reste de la division euclidienne.
- Calcul du PGCD de deux entiers.

Les points suivants feront l'objet d'une attention particulière lors de la réalisation du projet :

⋆ Clarté du programme.

⋆ Utilisation judicieuse de la récursivité.

⋆ Réutilisation des fonctions qui viennent d'être définies pour l'écriture de nouvelles fonctions.

Recherche exacte et approchée de mots dans un texte

Sujet proposé par Nadia El-Mabrouk

Introduction :

La recherche de toutes les occurrences d'un mot dans un texte est un problème très étudié et qui a des applications évidentes dans l'analyse d'un texte en français, la recherche bibliographique, etc.

Une autre application consiste en la recherche de motifs dans les séquences biologiques. L'ADN, qui constitue la carte génétique d'un individu, est formé d'une suite de 4 nucléotides notés A,C,G,T. On peut ainsi le visualiser comme un texte sur un alphabet de 4 lettres. Le long de la séquence d'ADN, certains motifs sont connus pour avoir un rôle particulier. Ces motifs peuvent être des mots ou des ensembles de mots, et se retrouvent de façon approximative dans le génome. Par exemple, "TAT(A ou G)AT" a été identifié comme étant un promoteur, c'est à dire un signal de démarrage de la transcription d'un groupe de gènes en protéine. Il existe également des signaux d'arrêt de la transcription à la fin du groupe de gènes. Le motif "GTTC*A**C", où * représente une lettre quelconque, est quant à lui un motif très conservé dans les gènes d'ARN de transfert.

Ainsi, la localisation de ce type de motifs dans le génome participe au décodage de l'ADN.

Projet :

Le but de ce projet est la recherche exacte ou approchée d'un mot ou d'une classe de mots dans un texte.

Un texte sera mémorisé dans un tableau de taille constante (par exemple 10000 caractères).

L'utilisateur devra avoir accès au menu suivant:

Saisie d'un nouveau texte	$\longrightarrow$	**(0)**
Recherche exacte d'un mot	$\longrightarrow$	**(1)**
Recherche approchée d'un mot	$\longrightarrow$	**(2)**
Recherche exacte d'une classe de mots	$\longrightarrow$	**(3)**
Recherche approchée d'une classe de mots	$\longrightarrow$	**(4)**

(0) Consiste à demander un mot, et à rechercher toutes les positions dans le texte où l'on a occurrence du mot.

Exemple1:
Mot recherché : masson
texte : La maison du masson est massive
On a une seule occurrence du mot à la position 14 dans le texte.

(1) Consiste à demander un mot et un nombre maximum d'erreurs (k), et à rechercher toutes les positions dans le texte où l'on a occurrence du mot à k erreurs près. On faira afficher la liste des mots approximatifs trouvés. Les erreurs considérées ici sont les substitutions de lettres.

Exemple2:
Le mot recherché et le texte sont les mêmes que dans Exemple1.
Soit $k = 1$

```
La maison du masson est massive
   masson
             masson
                        masson
```

A la position 4 , on a occurrence du mot avec une erreur : maison;
A la position 14, on a occurrence exacte du mot.
On a donc deux occurrences du mot dans le texte à k erreurs près, ou $k = 1$.

Si $k = 2$, on a une occurrence supplémentaire à la position 25 : massiv;

(3) On étend la notion de mot pour pouvoir tenir compte de classes de caractères, de complémentaires, de trous. Plus précisemment, chaque position du mot P pourra être:

- x: Un caractère de l'alphabet considéré;
- *: Ce caractère, qu'on appellera "trou", coïncide avec toutes les lettres de l'alphabet. Cela signifie qu'à cette position le mot peut contenir un caractère quelconque;
- [*caractères*]: Une classe de caractères. On autorise les intervalles (exemples : [a,e,i,o,u], [A..Z]);
- #C:Le complément d'un caractère ou d'une classe de caractères C (exemple : #[A..Z] représente la classe de tous les caractères moins les lettres majuscules).

On dira que P est une classe de mots.

Exemple3:
La classe de mots P=[Mm]a#[aeiou]*#a[a..p] contient les mots "masson" et "massif", mais pas "maison" et "massiv".

Le travail consiste donc ici à demander une classe de mots P de la forme décrite ci_dessus, et de rechercher dans le texte toutes les positions où l'on a occurrence d'un mot de P.

(4) Consiste à demander une classe de mots P et un nombre maximum k d'erreurs, et à rechercher toutes les positions dans le texte où l'on a occurrence d'un mot de la classe à k erreurs près.

Exemple4:
Reprenons le texte de Exemple1 et la classe de mots P de Exemple3.
Prenons $k = 1$.
Comme "masson" est une occurrence exacte et que "maison" et "massiv" sont deux occurrences à une erreur près de P, on a donc trois occurrences de mots de P dans le texte aux positions respectives : 4 , 14 , 25.

Le jeu de la vie

sujet proposé par Franck SICARD

Introduction

Le jeu de la vie se déroule normalement sur un quadrillage infini à mailles carrées (On travaillera sur un quadrillage fini).

Chaque maille du quadrillage contient soit une cellule vivante (notée par $\bullet$, dans les schémas suivants), soit une cellule morte (notée par $\circ$dans les schémas suivants).

On appelle voisines d'une cellule les 8 cellules qui sont en contact avec celle-ci. Ainsi on constate que la cellule $\star$ de l'exemple ci-dessous possède pour voisines 3 cellules vivantes ($\bullet$) et 5 cellules mortes ($\circ$).

$$\begin{array}{ccccc}
\circ & \bullet & \circ & \circ & \bullet \\
\circ & \bullet & \circ & \circ & \circ \\
\circ & \bullet & \star & \circ & \circ \\
\bullet & \circ & \circ & \bullet & \circ \\
\bullet & \circ & \bullet & \circ & \circ
\end{array}$$

On détermine l'état du plateau de jeu au tour suivant en appliquant les régles suivantes:

- Une cellule vivante meurt d'asphyxie si elle possède moins de 2 voisines.
- Une cellule vivante meurt d'étouffement si elle possède plus de 3 voisines.
- Une cellule morte ressucite si elle possède exactement 3 voisines.
- Dans tous les autres cas il n'y a pas de changement.

En itérant ce processus plusieurs fois, on constate l'apparition des phénomènes suivants:

- Il y a des formes stables qui ne bougent pas: dans l'exemple suivant chacune des 4 cellules vivantes possède exactement 2 voisines, donc elles vivent. Et aucune des cellules mortes possède exactement 3 voisines pour ressuciter. Donc la configuration ne se transforme pas.

$$\begin{array}{ccccc}
\circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \bullet & \circ & \bullet & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ
\end{array}$$

- Il y a des formes oscillantes immobiles:

$$
\begin{array}{ccccc}
\circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ
\end{array}
\Longrightarrow
\begin{array}{ccccc}
\circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ \\
\circ & \bullet & \bullet & \bullet & \circ \\
\circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ
\end{array}
\Longrightarrow
\begin{array}{ccccc}
\circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ
\end{array}
$$

- Il y a des formes oscillantes mobiles:

$$
\begin{array}{cccccc}
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ & \circ \\
\circ & \circ & \circ & \bullet & \circ & \circ \\
\circ & \bullet & \bullet & \bullet & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ
\end{array}
\Longrightarrow
\begin{array}{cccccc}
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \bullet & \circ & \bullet & \circ & \circ \\
\circ & \circ & \bullet & \bullet & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ
\end{array}
\Longrightarrow
$$

$$
\begin{array}{cccccc}
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \bullet & \circ & \circ \\
\circ & \bullet & \circ & \bullet & \circ & \circ \\
\circ & \circ & \bullet & \bullet & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ
\end{array}
\Longrightarrow
\begin{array}{cccccc}
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \bullet & \circ & \circ & \circ \\
\circ & \circ & \circ & \bullet & \bullet & \circ \\
\circ & \circ & \bullet & \bullet & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ
\end{array}
\Longrightarrow
$$

$$
\begin{array}{cccccc}
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \bullet & \circ & \circ \\
\circ & \circ & \circ & \circ & \bullet & \circ \\
\circ & \circ & \bullet & \bullet & \bullet & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ
\end{array}
$$

- Et bien d'autres formes plus complexes. . .

Implémentation

Un quadrillage infini étant non implémentable sur une machine, on se contentera d'un plateau fini de taille $N \times N$ repésenté par un tableau de booléens à 2 dimensions (**TRUE**=cellule vivante, **FALSE**=cellule morte):

```
...
int       Plateau [N][N] ;
```

De plus on supposera que le pourtour du plateau est constitué de cellules qui seront toujours mortes.

Une execution du programme devra se derouler comme suit:

1. Au départ le plateau est initialisé par uniquement des cellule mortes.
2. Saisir la liste des cellules vivantes en précisant leurs abscisses et leurs ordonnées.
3. Saisir le nombre d'itérations que l'on veut effectuer.
4. Indiquer si l'on veut afficher le resultat des itérations intermédiaires. Et si *oui* le programme attendra, par la suite, l'appui sur **RETURN** entre chaque affichage.
5. la simulation est effectuée en fonction des paramètres précédents, et le résultat est affiché à l'écran.

L'affichage se fera soit en mode texte par l'utilisation d'un *espace* pour représenter les cellules mortes et par un X pour représenter les cellules vivantes, soit en appliquant le TP sur le graphisme.
Toute fonctionnalité supplémentaire sera la bienvenue (par exemple la gestion d'un plateau circulaire (quand on atteind une extrémité du plateau, on se retrouve à l'autre extrémité)). On pourra également saisir l'emplacement des cellules vivantes initiales grâce à la souris

Représentation d'un échiquier

Marc Zipstein
zipstein@univ-mlv.fr

Le but du projet est de représenter un échiquier et les déplacements de pièces en cours de partie.

A chaque déplacement de pièce, le programme vérifie si le déplacement est valide et dans ce cas, affiche l'échiquier obtenu après le déplacement.

On pourra représenter l'échiquier et les pièces à l'aide de caractères alphabétiques (minuscules pour un camp et majuscules pour l'autre) et afficher des suites de tableaux successifs, ou, ce qui est beaucoup mieux, utiliser les fonctions graphiques vues en TP.
Les cases de l'échiquier seront repérées de la manière standard, lettre de a à h pour la colonne, chiffre de 1 à 8 pour la ligne.

Le programme devra effectuer la saisie alternée des déplacements pour chaque camp. Le déplacement sera indiqué par l'utilisateur sous la forme :

case de départ-case d'arrivée.

On devra pouvoir demander un affichage des déplacements possibles d'une pièce. La question sera indiquée par l'utilisateur sous la forme :

position suivie d'un point d'interrogation

Le programme devra permettre d'effectuer :

- *une partie normale* : pièces en position prédéfinie.
- *un problème* : pièces en jeu et positions des pièces précisées par l'utilisateur (on vérifiera la validité du problème : 1 seul roi par camp, au plus un fou de chaque couleur par camp, ...')

Il n'est pas demandé d'écrire un programme "jouant" aux échecs!

Billard graphique

Serge Riazanoff

Problème

On considère un billard dont les dimensions en largeur et hauteur peuvent être choisies par l'utilisateur. Le projet consiste à représenter la trajectoire d'une boule après un nombre variable de bandes (rebonds). Ce nombre est à entrer par l'utilisateur.

La position initiale de la boule et l'angle de frappe seront également fournis par l'utilisateur. La boule est supposé avoir été frappée sans effet. L'angle d'incidence est donc égal à celui de réflexion.

Le projet

On pourra utiliser pour le tracé des droites, de la boule ainsi que du cadre les outils de la librairie graphique présentés lors du T.P.

Il sera tenu compte des éléments suivants :

- qualité, commentaires et modularité de la programmation;
- contrôle des données saisies par l'utilisateur qui peuvent être quelconques ;
- aide et documentation pour l'utilisateur.

Bien entendu, on pourra compléter le projet en ajoutant par exemple d'autres obstacles (boules, champignons, etc...) ou en faisant figurer la trajectoire en pointillés.

Gestion de listes

Marc Zipstein

Préliminaires

Une liste est une suite ordonnée d'éléments. Une manière de représenter des listes en mémoire est de les conserver dans un tableau. Ainsi la liste $(5, 2, 6)$ pourrait être représentée par le tableau :

5	2	6

L'inconvénient d'une telle représentation est de nécessiter un grand nombre de déplacements d'objets lors de l'insertion ou de la suppression d'un élément. Ainsi, l'insertion en deuxième position du chiffre 9 dans la liste précédente nécessite le déplacement de deux éléments (2 et 6) :

5	9	2	6

Pour éviter cet inconvénient, on range dans le tableau des structures à deux champs ;

- le premier contient l'élément de la liste
- le deuxième l'indice du tableau où l'on trouve l'élément suivant.

Une telle structure est appelée **liste chaînée**. Dans notre exemple précédent la liste initiale serait :

indice	0	1	2
valeur	5	2	6
suivant	1	2	FIN

et après insertion en deuxième position du chiffre 9 :

indice	0	1	2	3
valeur	5	2	6	9
suivant	3	2	FIN	1

FIN est une valeur entière utilisée comme marqueur de fin de liste. Pour repérer les cases libres du tableau le programme doit gérer une liste chaînée des cases libres.

Problème

On écrira les fonctions permettant

- la création d'une liste d'entiers,
- l'insertion dans une liste à une position donnée,
- la suppression d'un élément,
- l'affichage à l'écran d'une liste ordonnée.

Le programme devra être capable de gérer plusieurs listes dans le même tableau.
Exemple d'un tableau contenant deux listes :

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} 4 & . & 7 & 6 & 2 & 3 & . & . & . & . & . & . \\ 3 & 6 & FIN & 4 & FIN & 2 & 7 & 8 & 9 & 10 & 11 & FIN \end{array}$$

liste $A = (4, 6, 2)$, liste $B = (3, 7)$ avec $r(A) = 0$, $r(B) = 5$, $r(V) = 1$ (liste des cases vides).

Bataille navale

Sujet proposé par Marc Zipstein

Le but du projet est de réaliser un programme permettant de jouer à la bataille navale. La partie minimum du projet consiste uniquement à gérer les réponses de l'adversaire.

Les différentes choix de l'utilisateur seront gérés à l'aide d'un menu. Par défaut, le terrain de jeu est un carré de 20∗20 cases et l'ensemble des bateaux est formé de :

Premier choix	1 navire de 6 cases,
Deuxième choix	2 navires de 5 cases,
Troisième choix	3 navires de 4 cases,
Quatrième choix	4 navires de 3 cases,
Cinquième choix	5 navires de 2 cases,
Sixième choix	6 navires de 1 case.

Une option du menu devra permettre à l'utilisateur de changer ces valeurs.

Déroulement du programme dans sa version minimale:

- première phase : définir la place des bateaux.
  Pour chaque nouveau bateau, on affichera un plateau de jeu avec les bateaux positionnés. Les bateaux ne peuvent être qu'en position horizontale ou verticale. Toutes les cases d'un bateau sont alignés et doivent être sur le plateau de jeu. Deux bateaux ne peuvent être situés sur des cases contigues. Tous les bateaux doivent être placés.
- deuxième phase : simulation de l'adversaire.
  Après avoir caché le plateau contenant les bateaux défini dans la première phase, on affiche un nouveau plateau indiquant la position et le résultat des tirs effectués, ainsi que les vaisseaux coulés.

Un tir est entré au clavier comme un couple de coordonnées dont on vérifiera la validité.

Le résultat indiqué sera :

- si on atteint un bateau :
  - *touché* si le bateau possède encore des cases non atteintes
  - *coulé* si toutes les cases du navire sont atteintes
- si on n'atteint pas de bateau:
  - *rien* s'il n'y a pas de bateau dans l'une des 8 cases adjacentes
  - *en vue* si au moins un navire est dans une des 8 cases adjacentes

Déroulement du jeu à deux joueurs

1. L'utilisateur place lui même ses bateaux.
2. Il dispose d'un plateau tir affichant les résultats de ses tirs et d'un plateau bateaux affichant ses bateaux et les tirs de l'adversaire.
3. Chaque joueur effectue un tir alternativement.

Une symbolisation claire sera choisie pour chacun de ces résultats.

Une option du menu devra permettre d'afficher simultanéement les deux plateaux, afin de permettre une vérification lors de la soutenance.

La représentation des vaisseaux pourra utiliser les caractères alphanumérique ou, beaucoup mieux, les fonctions de la bibliothèque graphique vues en TP.

Evolution du programme
Le jeu à deux joueurs:

La première évolution: consiste à placer aléatoirement les bateaux.

Pour cela on utilisera la fonction de hasard rand()qui renvoie un entier pseudo-aléatoire compris entre 0 et 32767. Afin de ne pas obtenir toujours les mêmes suites pseudo-aléatoires, la fonction doit être initialisée au début de votre programme, par l'instruction `srand(getpid())`. *Voir man rand.*

Deuxième évolution; Le jeu à un joueur, le deuxième étant géré par le programme.
Le menu propose plusieurs adversaires possibles, adoptant chacun des stratégies de jeu différentes.

Troisième évolution : le programme gère les deux joueurs afin de déterminer la meilleure des stratégie implémentées.

Quatrième évolution : suivant votre inspiration.

REVERSI
Jeu à deux joueurs

Sujet proposé par Frédéric Vielle

Description du jeu

0.0.1 Le matériel

- Un plateau de 64 cases (8 x 8).
- 64 pions ayant chacun une face blanche et une face noire.

Les règles du jeu

Au départ, 4 pions (2 sur la face noire et 2 sur la blanche) sont disposés en croix au centre du plateau. Les adversaires jouent à tour de rôle, sauf si un joueur se trouve dans l'impossibilité de placer un pion. Dans ce cas, il passe son tour. La partie prend fin lorsque tous les pions ont été joués ou lorsqu'aucun des deux joueurs ne peut plus jouer.

Chaque joueur joue avec une couleur. Les pions dont la face blanche (resp. noire) est visible seront appelés indifféremment pions blancs (noirs) ou pions du joueur blanc (noir) par la suite.

Pour jouer, le joueur doit placer un nouveau pion de sa couleur de telle sorte que :

- il soit aligné avec au moins un autre pion de sa couleur,
- entre ces deux pions, il n'y ait que des pions de l'autre couleur.

Il retourne alors les pions encadrés par le pion qu'il vient de jouer et ses autres pions.

Voici le début d'une partie. La figure 1 représente la position initiale et les figures 2, 3 et 4 le résultat de trois coups successibles possibles. Les deux couleurs sont symbolisées par o et x. o commence.

			o	x			
			x	o			

Fig 1

			o	o	o		
			x	o			

Fig 2

					x		
			o	x	o		
			x	o			

Fig 3

					x		
			o	x	o		
		o	o	o			

Fig 4

Dans les figure suivantes, les coups possibles pour o sont montrés à l'aide du symbole s et de p pour x.

					s		
				s	x		
			o	x	o		
		o	x	o			
		x	s				
		s					

Fig 5

			p		x		
		p	o	x	o	p	
	p	o	x	o	p		
		o		p			
	p	o					

Fig 6

					x		
			o	x	x		
		o	x	x	x		
		o					
		o					

Fig 7

But du jeu

Le joueur qui a le plus de pions de sa couleur à la fin de la partie a gagné.

Description du projet

Le projet consiste :

1. à fournir les outils pour pouvoir jouer :

    - dessin du plateau de jeu,
    - dessin des pions que l'on pourra placer et retourner à tour de rôle selon les règles énoncées ci-dessus (le programme doit rejeter les coups illégaux et retourner lui-même les pions après qu'un joueur aura joué),
    - le programme doit détecter la fin de la partie quand elle survient, ainsi qu'une situation où un joueur est obligé de passer.

2. à pouvoir indiquer au joueur les coups légaux possibles. On pourra essayer de déterminer le meilleur coup possible (en précisant selon quels critères).

Implémentation

Pour saisir le jeu des pions, vous pourrez numéroter les colonnes et mettre des lettres sur les lignes. Ainsi, chaque case aura un nom : (C-5), (E-3), ...

Le plateau pourra être représenté dans votre programme par un tableau d'entiers. Une case sera *1* si elle est occupée par un pion noir, *-1* ou si c'est un pion blanc, *0* dans le cas d'une case vide (par exemple). Voici donc une déclaration du plateau :

```
int Plateau[NB_LIGNE]{NB_COLONNE] ;
```

Les déplacements pourront aussi être indiqués à l'aide de la souris. Le jeu et les pions seront dessinés à l'aide de la bibliothèque graphique.

Amélioration

Donner la possibilité de faire jouer l'ordinateur. On pourra s'inspirer de la méthode suivante en remarquant que les cases de coin et les cases y menant sont importantes.

Pour choisir le meilleur déplacement du pion, vous pouvez noter les différentes positions possibles.

Exemple de notation :
- une case occupée ou noire aura une note négative (-1),
- les autres auront comme note le nombre de cases blanches vides adjacentes à la case à noter.

Le choix du programme ira vers la case ayant la plus grande note. Mais cette notation n'est pas forcément la meilleure. Peut-être trouverez-vous mieux ?

Pour interdire une case, vous pourrez utiliser le même principe. Une fonction évaluera la case la plus gênante pour l'adversaire et si possible la moins gênante pour soi. A vous de trouver une fonction approchant ces objectifs.

Représentation d'un labyrinthe

sujet proposé par Marc Zipstein

But

Le but du projet est de représenter une vue subjective des couloirs composants un labyrinthe, ainsi que de permettre un déplacement dans ce labyrinthe. Les déplacements se feront à l'aide d'une zone de 6 boutons :

- quart de tour à gauche,
- avancer d'une case,
- quart de tour à droite,
- pas sur le coté à gauche,
- pas en arrière,
- pas sur le coté à droite.

Description des fonctionnalités obligatoire

Le projet consiste en:

- affichage des lieux en fonction de la position du joueur dans le labyrinthe et de son orientation,
- gestion du mouvement, en contrôlant sa validité(pas de passe-muraille),
- détermination de la fin de partie (quand le joueur a atteint la sortie).

Implémentation

Le labyrinthe est contenu dans un rectangle de moins de LONGUEUR×LARGEUR cases.
Le labyrinthe sera stockée sous forme d' un fichier texte lu par le programme (les fonctions de manipulations de fichier sont fournies en annexe).

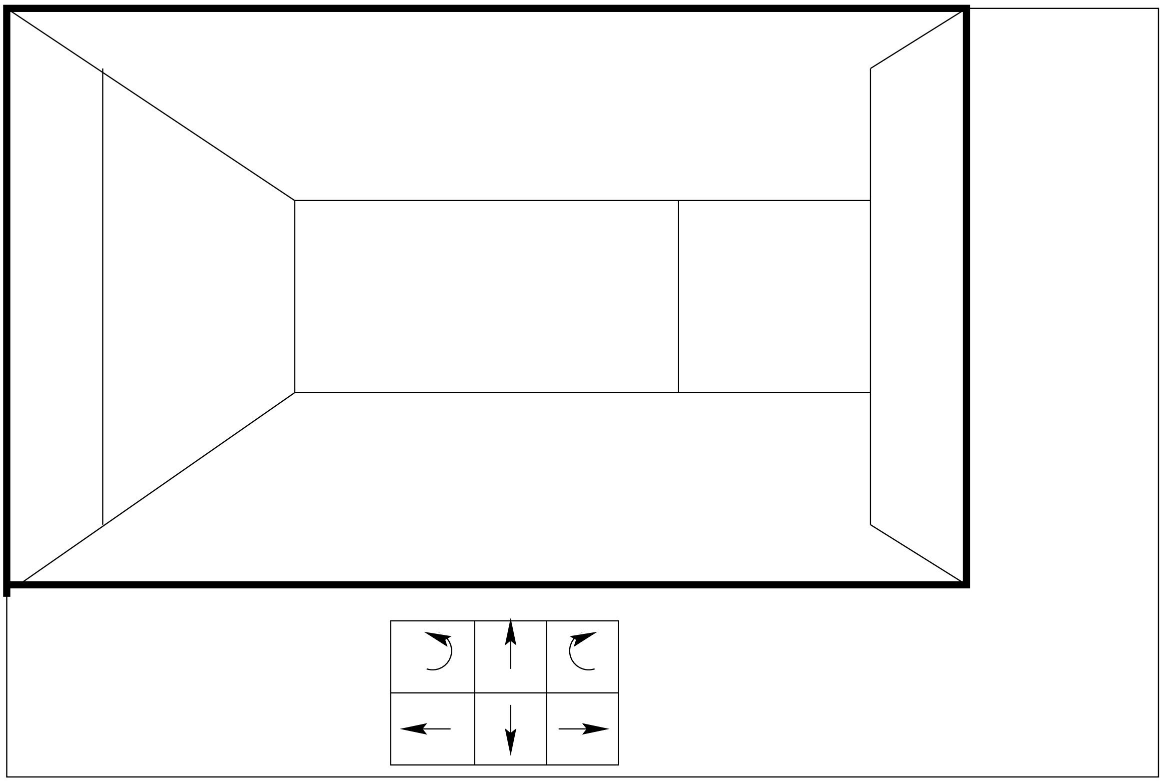
Les éléments du labyrinthe sont représenté dans le fichier texte par les caractères suivants:

- `*` pour un mur
- `.` pour une case vide
- `@` pour le joueur au début de la partie
- `x` pour la sortie

L'affichage en vue subjective se fera en '*fil de fer*' à une case de distance. Ainsi la partie du fichier texte :

```
******
*.....
*@****
*.****
```

avec une orientation vers le nord devra correspondre à l'affichage d'un tournant vers la droite. Une présentation possible de cette situation correspond à l'écran:



On devra gérer des salles, c'est à dire des murs éloignés de plus d'une case.

Améliorations possibles

- Affichage d'une carte construite au fur et a mesure du déplacement
- Possibilité de sauvegarde
- Gestion d'étages
- Génération aléatoire du labyrinthe
- Gestion d'habitants et de leur rencontre.

Annexe Gestion de fichier. Les fichiers sont manipulés grace à des objets de type `FILE *`.

un tel objet est associé à un fichier de nom `reference` par la fonction `fopen`. Il faut préciser le mode d'ouverture en transmettant une chaine de caractère :
"r" pour read -ouverture en lecture ,
"w" pour write -ouverture en écriture.
une bonne habitude est de tester si l'opération a réussi en testant le code de retour de la fonction. En effet `fopen` renvoie `NULL` en cas d'echec à l'ouverture (problème de droits par exemple).
Exemple
ouverture en lecture du fichier `labyrinthe1` situé dans le répertoire courant :

```
FILE *fich;
...
fich=fopen("labyrinthe1","r");
if (fich==NULL)
{
printf("ouverture impossible \n");
...
}
else
{ ....
```

Les fonctions d'entrée sortie formatées sont `fprintf` et `fscanf`. Leur syntaxe est identique à celle de `printf` et `scanf` si ce n'est qu'il faut fournir un `FILE *` convenablement initialisé avant le format. Ainsi la lecture du caractère suivant du fichier associé à `fich` est obtenue par :

```
fscanf(fich,"\%c",&car)}
```

Trois En Ligne

Sujet proposé par Marc Zipstein

Le plateau de jeu est un carré constitué de COTE$\times$ COTE cases de taille identique. Chaque case est remplie par une figure choisie au hasard. Sur le plateau, il y a au plus NBFIGURES différentes.

L'objectif du jeu consiste à éliminer un maximum de figures.
Pour cela le joueur disopse de NBTOURS. A chaque tour, le joueur peut déplacer une figure en l'échangeant avec une figure voisine (une figure voisine est adjacente par un de ses cotés).Cet échange est réalisé en cliquant successivement sur la première figure puis sur sa voisine.
A l'issu de cette opération, les cases qui sont élinimés sont celles qui vérifient les règles suivantes :

- **version minimum**
  Si, après déplacement de la première cellule, elle appartient à une ligne ou une colonne de plus de 3 figures adjacentes identiques, ces figures sont éliminées et sont remplacées par les figures situées au dessus. Lorsqu'il n'y a plus de figures au-dessus, de nouvelles figures sont crées aléatoirement pour remplir les cases vides.

- **version plus évoluée**
  Tant que le plateau contient des lignes ou des colonnes de plus de trois figures identiques, celles-ci sont éliminées avant l'échange suivant.

Pour toutes les versions, la fenêtre graphique comprendra :

- le plateau de jeu;
- un compteur de tour;
- un compteur de nombre de figures éliminées;
- pour la deuxième version un indicateur informera l'utilisateur qu'un clic est attendu (l'élimination répétée des lignes et colonnes de plus de 3 éléments prend un certain temps).

On pourra choisir les valeurs :

```
#define COTE 10
#define NBFIGURES 5
#define NBTOURS 10
```