Expressions régulières

- But: parcourir un fichier et extraire les lignes qui satisfont une expression régulière donnée (programme *grep* sous Unix)
- Wildcards:
  - `.` caractère quelconque
  - `*` le caractère précédent est répété 0 ou *n* fois
  - `[]` liste de caractères à choix
  - `[^]` négation
  - `^` début de la ligne
  - `$` fin de la ligne
- Exemple

  mots: tata, toto, tota patterns: …a [oa] t[^o]

Algorithmes de tri I

- But: implémenter différents algorithmes de tri génériques et comparer leurs performances en fonction de la taille des tableaux à trier
- Inspirez-vous du prototype de la fonction qsort()

```
void qsort(
        void *base,
        size_t num,
        size_t width,
        int (*compare )(const void *, const void *)
    );
```

- Testez votre fonction avec différents types de tableaux: entiers, chaines de caractères, rationnels et pointeurs sur des rationnels
- Algorithmes:
  - Insertion sort
  - Bubble sort
  - Merge sort
  - Quicksort
- Comparer avec les implémentations existantes
  - qsort
  - std::sort

Algorithme de tri II

- But: visualiser le comportement d'algorithmes de tri

- Implémentez plusieurs algorithmes pour trier des tableaux d'entiers, et affichez leur progression
  - Bubble sort, bubble sort bidirectionnel, merge sort

  Screenshot:

  http://java.sun.com/docs/books/tutorial/essential/threads/

Inversion matricielle

- But: implémenter des opérations sur des matrices

$$A^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} = \frac{1}{|A|} \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix}$$

- Inversion par bloc

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}$$

- Elimination de Gauss-Jordan

  http://mathworld.wolfram.com/Gauss-JordanElimination.html

- Décomposition LU

  http://mathworld.wolfram.com/LUDecomposition.html

Elimination de Gauss-Jordan

- Construire la matrice augmentée

  [A I]

- Effectuer une élimination de Gauss (une suite d'opérations élémentaires sur les lignes) pour obtenir la matrice

  [I A^{-1}]

  http://mathworld.wolfram.com/GaussianElimination.html

- Utiliser le pivotage pour garantir la stabilité numérique

Décomposition LU

- Décompose une matrice A en un produit de matrices triangulaires inférieures et supérieures

$$A = LU, \qquad \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{12} & l_{22} & 0 \\ l_{13} & l_{23} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

- Algorithme de Doolittle

  $A^{(n)}=L_nA^{(n-1)}$ $A^{(0)}=A$,

  $U=A^{(n-1)}$ $L=L_1^{-1}\dots L_{N-1}^{-1}$

  http://en.wikipedia.org/wiki/LU_factorization

$$L_n = \begin{pmatrix} 1 & & & & 0 \\ & \ddots & & & \\ & & 1 & & \\ & & l_{n+1,n} & \ddots & \\ & & \vdots & & \ddots \\ 0 & & l_{N,n} & & 1 \end{pmatrix}. \qquad l_{i,n} := -\frac{a_{i,n}^{(n-1)}}{a_{n,n}^{(n-1)}}$$

- En utilisant la substitution inverse, inverser U et L pour obtenir l'inverse de A

  $A^{-1}=(LU)^{-1}=U^{-1}L^{-1}$

Transformations linéaires

- But: effectuer rotations, translations et homothétie sur des images, en contrôlant les opérations à l'aide du clavier

- Exprimer les opérations sous forme de matrices homogènes, et effectuer les multiplications correspondantes pour trouver l'emplacement de chaque pixel de l'image d'arrivée dans l'image de départ
- Améliorez les performances en utilisant une approche incrémentale
- Variez les incréments pour produire des déformations non linéaires

malloc & free

- But: Réimplémenter malloc et free
  - Demander de la mémoire au système
  - Chaque bloc a un en-tête définissant sa taille et l'adresse du bloc suivant, formant un liste chainée de blocs libres
  - Lorsque malloc est appelé, on partitionne un bloc suffisamment grand parmi ceux les blocs disponibles, ou si aucun bloc n'est suffisamment grand, on demande à nouveau de la mémoire au système
  - Lorsque free est appelé, on réinsère le bloc libéré dans la liste
- Ecrivez un programme de test pour vérifier la qualité de votre implémentation
- Comparez avec le malloc officiel
- Mesurer la consommation totale de mémoire en fonction de la stratégie d'allocation utilisée
  - First fit: partitionner le premier bloc suffisamment grand
  - Best fit: partitionner le bloc dont la taille est la plus proche de la taille demandée
  - Worst fit: partitionner le bloc dont la taille est la plus grande