Fondue dauphinoise

Projet Programmation C, DEMI2E 2018/2019

1 Informations importantes

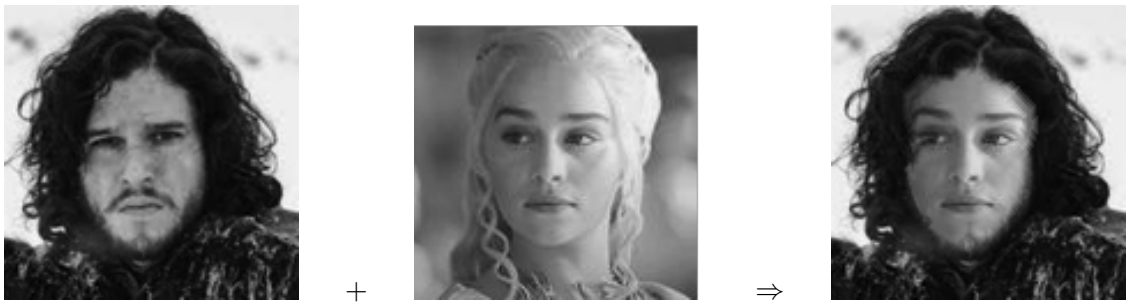
- Deadline pour la composition des binômes sur le Wiki de MyCourse : mercredi 26 décembre 2018, 23h.
- Rendu du projet : A déterminer, probablement début février.
- Soutenances : à fixer
- Espace Piazza pour les questions/réponses : https://piazza.com/class/jpv7zmhz1zb5a5

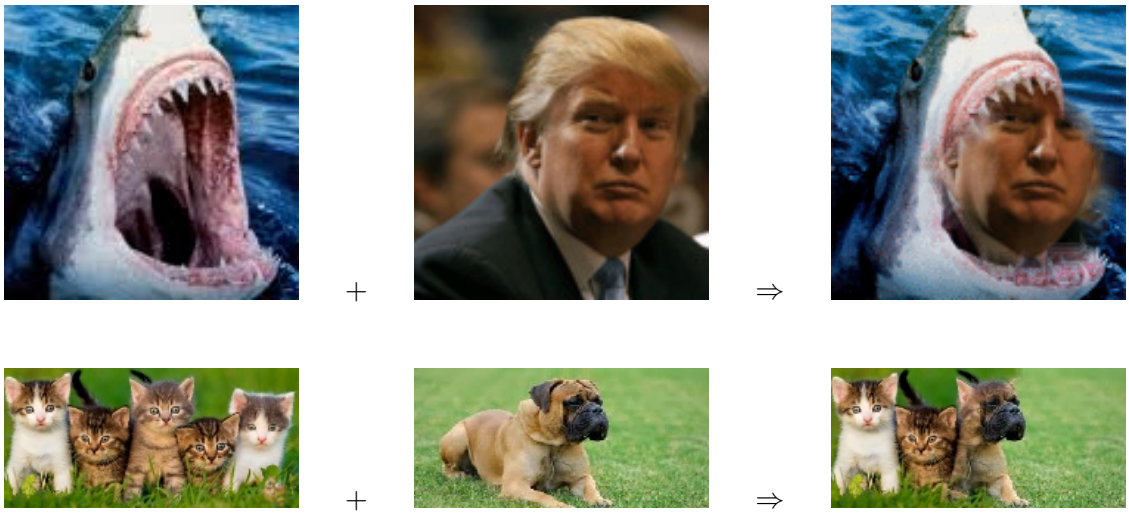
Versions Ce sujet est susceptible d'être modifié, veillez à travailler avec la dernière version située sur MyCourse.

Version actuelle : version du 21 décembre 2018.

2 Description

L'objectif de ce projet est d'écrire un programme permettant de mélanger automatiquement deux images de même taille entres elles, de telle façon qu'elles aient l'air d'avoir été fondues ensemble. Voici quelques exemples :





3 Principe

Pour fusionner deux images I_1 et I_2 en une troisième image I_3 de même taille, il suffit de définir chaque pixel de I_3 comme étant un pixel soit de I_1 soit de I_2. Cela revient à placer les pixels de I_3 dans deux sous-ensembles S et T où S correspond aux pixels issus de I_1 et T correspond à ceux issus de I_2. S et T doivent être disjoints et leur union doit être l'ensemble des pixels de I_3. On parle alors de *partition* des pixels. Cependant, pour obtenir un effet de fondu comme dans les images présentées précédemment, on souhaiterait que les pixels de S ou T soient contigus dans I_3.

Ce problème de partitionnement des pixels en deux catégories contigües peut se ramener à un problème classique de la *théorie des graphes*. Un *graphe* est une représentation d'un problème sous la forme d'un ensemble de *sommets* reliés par un ensemble d'*arêtes* représentant les connexions entre les sommets. Le réseau routier par exemple est représenté par un graphe dans lequel les sommets sont les villes et les arêtes sont les routes reliant les villes. Une arête entre deux villes existe dans le graphe s'il existe une route qui les relie.

Pour le problème traité dans ce projet, une image va être représentée par un *graphe de voisinage*. Chaque pixel de l'image est un sommet du graphe, et il existe une arête reliant deux sommets du graphe si les deux pixels correspondants sont voisins dans l'image (la plupart des pixels ont 8 voisins).

Le problème de la fusion de deux images revient alors à partitionner les sommets du graphe en deux sous-ensembles disjoints, ce qui peut se ramener à déterminer une coupe de capacité minimale dans le graphe. En effet, une *coupe* dans un graphe est une partition des sommets en deux sous-ensembles disjoints.

L'idée de ce projet est donc de construire un graphe (voir section 4) à partir des deux images que l'on souhaite mélanger de telle sorte que trouver l'endroit où les couper/coller corresponde à résoudre le problème de trouver une coupe de faible capacité (voir section 6) dans ce graphe.

4 Graphes d'images

Pour ce projet, on modélise le voisinage des pixels des deux images à mélanger à l'aide d'un seul graphe. Pour que cela puisse fonctionner, les images doivent avoir exactement les mêmes dimensions en nombre de pixels.

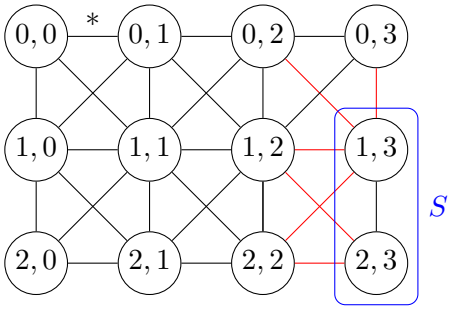
À chaque couple de pixels de mêmes coordonnées dans les deux images, on fait correspondre un sommet du graphe. Chaque sommet est relié à tous les sommets qui correspondent à des pixels voisins dans l'image (en ligne, en colonne et en diagonale). Par exemple :

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

img1

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

img2



Graphe correspondant

On associe une valeur aux arêtes. Pour un graphe routier par exemple, cette valeur peut représenter la distance entre deux villes reliées par une route. Dans ce projet, la valeur associée à une arête reliant le sommet (i_1, j_1) au sommet (i_2, j_2) correspond au coût de faire appartenir le pixel (i_1, j_1) et le pixel (i_2, j_2) à des images différentes. Cette valeur est appelée *capacité*. Elle est calculée comme une combinaison des valeurs RGB des quatre pixels qui correspondent aux deux extrémités [1] de l'arête. Dans notre exemple, la capacité de l'arête marquée avec * est : `Capa`(img1[0,0], img2[0,0], img1[0,1], img2[0,1]), où `Capa` est la fonction décrite dans la section 7.

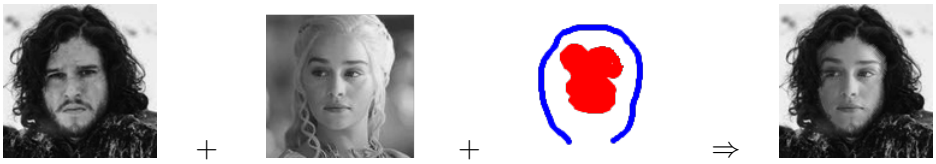
5 Coupe pour la fusion d'images

Coupe Une *coupe* est une partition des sommets d'un graphe en deux sous-ensembles disjoints S et T. On appelle également coupe l'ensemble des arêtes ayant une extrémité dans chaque sous-ensemble de cette partition, i.e. reliant un sommet de S à un sommet de T. Dans l'exemple de la section 4, on a l'ensemble $S = \{(1,3),(2,3)\}$ (et donc T contient tous les autres sommets) et la coupe peut être représentée par l'ensemble des arêtes dessinées en rouge. La capacité d'une coupe est la somme des capacités de ses arêtes.

La recherche de l'endroit où la fusion de deux images peut sembler la plus naturelle possible peut donc se ramener à déterminer une coupe de capacité minimale dans le graphe d'images construit dans la section 4.

1. Les deux sommets reliés par une arête sont appelés les *extrémités* de cette arête.

Initialisation de la coupe pour la fusion d'image Une coupe du graphe d'images définie par les deux ensembles S et T indique donc comment construire la fusion de deux images I_1 et I_2 : les pixels correspondant aux sommets de S sont ceux qui seront issus de I_1, et ceux correspondant au sommet de T seront issus de I_2. On souhaiterait éviter d'obtenir une coupe dans laquelle l'un des deux ensembles S ou T contiendrait un seul ou un nombre très faible de sommets. Pour cela, nous allons donc imposer à certains sommets d'appartenir à S et d'autres à T, en initialisant S et T avec ces sommets. Le choix des sommets (i.e. pixels) à imposer va dépendre du type d'images que l'on souhaite fusionner. Ainsi, pour un rendu comme celui des chiens et chats, il suffit de prendre les bords verticaux de l'image (ceux de gauche doivent être placés dans S, et ceux de droite dans T par exemple). En revanche, pour des effets comme ceux des différents visages, il faut des ensembles de pixels qui correspondent plus ou moins aux contours. Le plus simple est de fabriquer une troisième image (que l'on appelle un *masque*) de même taille, indiquant quels pixels doivent être placés dans S (ceux coloriés en bleu dans le masque) et lesquels doivent être placés dans T (ceux coloriés en rouge dans le masque). Les pixels non coloriés iront soit dans S soit dans T :



6 Détermination d'une coupe de faible capacité

Dans ce projet, il est demandé de mettre en oeuvre des méthodes heuristiques pour déterminer une coupe de faible capacité dans un graphe d'images. Pour chacune des méthodes, nous supposons que nous disposons des ensemble S et T initiaux **non vides**. Il peuvent avoir été initialisés en choisissant aléatoirement un sommet par exemple, ou par un ensemble de sommets précis (masque ou bords de l'image).

Parcours en largeur Une méthode naïve consisterait à parcourir en largeur le graphe à partir d'un sommet de S, jusqu'à ce qu'une condition d'arrêt soit vérifiée. Les sommets non encore atteints à l'issue du parcours peuvent alors être placés dans T. Exemples de conditions d'arrêt : dès qu'un sommet de T est rencontré (si T n'est pas vide), après X itérations, si le parcours ne permet plus d'améliorer la capacité de la coupe courante,... Dans cette méthode, S doit être initialisé avec au moins un sommet, mais T peut éventuellement être initialisé à l'ensemble vide.

On peut rappeler qu'un parcours en largeur dans un graphe s'effectue à l'aide d'une *file* contenant les prochains sommets à parcourir, de la manière suivante :

```
Mettre le ou les sommet(s) de S dans la file
Tant que la condition d'arrêt n'est pas vérifiée
   Retirer le sommet du début de la file //il est considéré comme parcouru
   Enfiler tous ses voisins non encore parcours
```

Algorithme de Karger L'algorithme de Karger [2] est un algorithme probabiliste simple déterminant une coupe contenant peu d'arêtes dans un graphe. Il repose sur la notion de *contraction* aléatoire d'arêtes. La contraction d'une arête (u, v) consiste à *fusionner* les sommets u et v en un seul sommet uv relié à la fois aux voisins de u et à ceux de v. Cette transformation peut faire apparaître des arêtes en double (ayant les mêmes extrémités) si u et v partagent un voisin. On parle alors de *multi-arêtes*.

La contraction d'une arête $e = (u, v)$, où u et v désigne ses deux extrémités, peut s'effectuer de la manière suivante :

```
Contraction de e=(u,v) :
   Ajouter tous les voisins de v à ceux de u
   Ajouter v à la liste des sommets représentés par u
   Supprimer v du graphe
   Remplacer toutes les occurrences de v dans le graphe par u
   Supprimer les arêtes de u à u
```

Soit un graphe $G = (V, E)$, où V désigne l'ensemble des sommets et E l'ensemble des arêtes du graphe, un pseudo-code de l'algorithme de Karger est :

```
Algorithme de Karger pour G=(V,E) :
Tant que |V| > 2
    choisir aléatoirement une arête e dans E
    contracter e dans G
Retourner l'unique multi-arête de G
```

À la fin de l'algorithme, il n'y a plus que deux sommets S et T dans le graphe, contenant l'ensemble des sommets fusionnés soit en S soit en T, ce qui constitue une partition des sommets du graphe initial. La multi-arête reliant S et T contient l'ensemble des arêtes de la coupe.

Comme c'est un algorithme probabiliste, il échoue souvent à donner la meilleure coupe. Cependant, sa probabilité d'erreur est bornée. Pour avoir la meilleure coupe possible, vous devrez relancer cet algorithme un très grand nombre de fois et reporter la meilleure des solutions.

Notons cependant que cet algorithme détermine une coupe contenant un nombre faible d'arêtes, et non une coupe de *faible capacité* construite à partir d'ensembles S et T *non vides*. Il faut donc l'adapter au problème traité dans ce projet en ajouter les traitements suivants :

- interdire la contraction d'une arête dont une extrémité appartient à S et l'autre à T.
- pondérer le choix aléatoire de l'arête à contracter par sa capacité. Ainsi, plus une arête a une capacité élevée plus elle a de chances d'être sélectionnée.

7 Détails techniques

Gestion des images et des couleurs Le projet devra manipuler des fichiers images au format PPM ASCII (P3) .

À la différences des images manipulées en TP, celles-ci sont en couleur. Par conséquent, chaque pixel est représenté par son niveau de Rouge, de Vert et de Bleu (RGB). Aussi, chaque

2. https://fr.wikipedia.org/wiki/Algorithme_de_Karger

ligne ne devant normalement faire pas plus de 70 caractères, certains convertisseurs respectent cette règle. Ainsi, à partir de la ligne 4, on trouve $3 \cdot h \cdot l$ valeurs, séparés aussi bien par des espaces que par des retours à la ligne. Des lignes commentaires (à ignorer) débutent par le caractère #.

Pour convertir n'importe quelle image en ppm, il existe des convertisseurs. Par exemple sous linux :

```
convert -compress none image.jpg image.ppm
```

Vous pouvez également utiliser le logiciel Gimp (disponible sous Windows, Linux et Mac). Il faut bien vérifier que les images obtenues avec ces outils sont au format `ppm` ASCII (P3) et non `ppm` binaire (P6).

Fonction de capacité La fonction de capacité, calculée entre deux sommets u et v, attribue un coût au fait de définir u avec l'image I_1 et v avec l'image I_2. Il est faible si u et v sont très différents dans (au moins) une des deux images, ou si les deux images sont très similaires sur ces deux pixels (dans le premier cas la coupe suivra la limite d'un objet, dans le second la transition se fera au niveau d'une zone uniforme). Pour calculer ce coût, on utilise une fonction (donnée) Δ, qui, étant donnés deux pixels, renvoie une mesure de la différence entre leurs couleurs entre 0 et 1. La fonction de capacité est alors

$$F(I_1(u), I_2(u), I_1(v), I_2(v)) = \frac{\Delta(I_1(u), I_2(u)) + \Delta(I_1(v), I_2(v))}{\Delta(I_1(u), I_1(v)) + \Delta(I_2(u), I_2(v)) + 10^{-4}}.$$

La fonction Δ est la suivante :

$$\Delta(P1, P2) = \frac{\sqrt{\frac{(R(P1)-R(P2))^2+(G(P1)-G(P2))^2+(B(P1)-B(P2))^2}{3}}}{255}$$

où $P1$ et $P2$ sont deux pixels et où R (resp. G et B) est une fonction donnant la valeur de la composante rouge (resp. vert et bleu) d'un pixel.

Exemples On fournit avec le sujet du projet les jeux d'exemples présentés au début du sujet, ainsi que le masque.

8 Travail minimum d'implémentation demandé

Implémentation des deux méthodes heuristiques Les différentes étapes sont :

1. Construction du graphe d'images à partir de deux images. La représentation en langage C du graphe est libre, mais il vous est demandé pour ce projet de représenter les voisins d'un sommet donné à travers la **liste des arêtes** qui lui sont adjacentes.
2. Initialisation des ensembles S et T, à partir des bords, du masque, ou aléatoirement selon la méthode et/ou les images testées.
3. Implémentation de l'heuristique de parcours en largeur du graphe. Vous pouvez tester plusieurs initialisations de S et T et plusieurs conditions d'arrêt. Vous pouvez aussi en proposer d'autres qui vous paraissent pertinentes. Vous pouvez lancer plusieurs fois l'algorithme et récupérer le meilleur résultat.
4. Construction de l'image résultat à partir de S et T.

5. Implémentation de l'algorithme de Karger. Vous pouvez implémenter une première version qui ne tient pas compte des capacités des arêtes, puis une version qui en tient compte. Vous pouvez comparer les résultats obtenus avec les deux versions.

Entrées-Sorties Votre programme doit pouvoir prendre en entrée deux images I_1 et I_2 aux formats `PPM ASCII` de même dimension, et éventuellement une troisième image masque. À partir de ces fichiers, il construit plusieurs images résultant de la fusion de I_1 et I_2 selon les différentes méthodes utilisées et leurs variantes. Il peut aussi afficher les valeurs des coupes obtenues. Si le masque n'est pas donné en argument, alors l'initialisation des ensembles S et T peut s'effectuer avec les bords, aléatoirement ou selon un masque par défaut. Si les images I_1 et I_2 n'ont pas les mêmes dimensions, le programme se contente d'afficher un message d'erreur.

Exemple attendu pour lancer votre projet :

```
./a.out image1.ppm image2.ppm masque.ppm
```

9 Améliorations

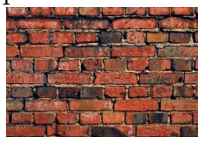
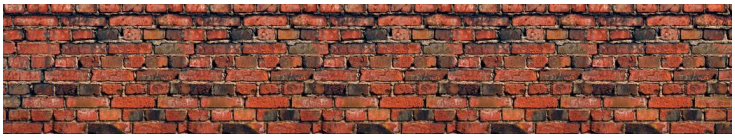
Une fois que le mélange d'images basique fonctionne, vous pouvez implémenter au moins l'une des améliorations suivantes, au choix.

Adoucir la coupe Pour améliorer le rendu du mélange d'images, une possibilité est d'essayer de "flouter" légèrement la zone autour de la coupe. Pour cela, il faut déterminer l'ensemble des sommets à distance d ou moins des sommets de la coupe, avec un d que vous choisirez en testant et appliquer le "flou" vu en TP.

Tester une autre heuristique Plutôt que d'effectuer un parcours en largeur du graphe, on peut appliquer une méthode de recherche locale qui consisterait à ajouter un ou plusieurs sommets à S (ou T) tant que cela améliore la coupe.

Textures L'exemple des briques ci-dessous est obtenu en répétant plusieurs fois une image qui peut facilement s'auto-recouvrir sur les bords. Pour cela, il faut pouvoir fabriquer des collages d'images et non plus des superpositions, en "collant" deux images I_1 et I_2 (qui sont deux copies de la même image) en une troisième image, plus large, où la partie gauche correspond à I_1 et la partie droite à I_2, avec une coupe au milieu. Les images ne sont pas simplement juxtaposés, la coupe n'est pas forcement droite mais va chercher à obtenir le meilleur collage possible. Vous pouvez proposer un mode d'exécution pour ce type de rendu.

Exemple :

5 x  ⇒ 

Améliorer la coupe Implémenter l'algorithme d'Edmonds-Karp donnant une coupe optimale. [3]

3. https://fr.wikipedia.org/wiki/Algorithme_d%27Edmonds-Karp

10 Conditions de rendu

Le projet est à effectuer en **binôme**, *i.e.* par 2 (**DEUX**) personnes. En cas de nombre impair d'élèves, **un seul** groupe sera autorisé à effectuer le projet en **trinôme** (notation plus sévère). Si au final il y a plus d'un trinôme ou plus de zéro monôme, les projets correspondants auront une note de 0. Pour former les groupes, utiliser le wiki mis à disposition sur l'espace MyCourse.

Le projet est à rendre sur l'espace MyCourse dédié avant la date et l'heure indiquées plus haut. **Chaque heure de retard sera pénalisée d'un point sur la note finale** (une heure entamée étant due). Une fois votre fichier envoyé, vous devez avoir un écran de confirmation avec votre fichier et la date de l'envoi. Le format de rendu est une archive au format ZIP contenant :

- Le code-source de votre projet (éventuellement organisé en sous-dossiers).
- Un répertoire *docs* contenant :
  - Une documentation pour l'utilisateur *user.pdf* décrivant à un utilisateur quelconque comment se servir de votre projet.
  - Une documentation pour le développeur *dev.pdf*, devant justifier les choix effectués, les avantages et inconvénients de vos choix, expliquer les algorithmes et leur complexité, indiquer quelles ont été les difficultés rencontrées au cours du projet ainsi que la répartition du travail entre les membres du binôme. Un programmeur averti devra être capable de faire évoluer facilement votre code grâce à sa lecture. En aucun cas on ne doit y trouver un copier/coller de votre code source. Ce rapport doit faire le point sur les fonctionnalités apportées, celles qui n'ont pas été faites (et expliquer pourquoi). Il ne doit pas paraphraser le code, mais doit rendre explicite ce que ne montre pas le code. Il doit montrer que le code produit a fait l'objet d'un travail réfléchi et minutieux (comment un bug a été résolu, comment la redondance dans le code a été évitée, comment telle difficulté technique a été contournée, quels ont été les choix, les pistes examinées ou abandonnées...). Ce rapport est le témoin de vos qualités scientifiques mais aussi littéraires (style, grammaire, orthographe, présentation).
  - Un court rapport d'expériences appelé *exp.pdf* où vous analyserez les résultats de vos tests, comparerez les différentes approches au niveau de leur pertinence etc.
- Optionnellement un makefile.

L'archive aura pour nom Nom1Nom2.zip, où Nom1 et Nom2 sont les noms des membres du binôme par ordre alphabétique (sans les accents ni d'éventuels espaces). L'extraction de l'archive devra créer un dossier Nom1Nom2 contenant les éléments précisés ci-dessus.

Il va sans dire que les différents points suivants doivent être pris en compte :

- Projet compilant sans erreur ni warning avec l'option -Wall de gcc et fonctionnant sur les machines de l'université (on vous conseille de compiler vos projets avec l'option `-O3` qui permet **une exécution plus rapide** de vos programmes).
- On préfère un projet fonctionnant parfaitement avec peu de fonctionnalités qu'un projet qui a tenté de répondre à tout mais mal.
- Vérification des données entrées par l'utilisateur et gestion des erreurs.
- Uniformité de la langue utilisée dans le code (anglais conseillé) et des conventions de nommage et de code.

– La documentation ne doit pas être un copié-collé du code source du projet.
– Les sources doivent être commentées, dans une unique langue, de manière pertinente (pas de commentaire “fait un test” avant un if.).
– Les noms des variables et fonctions doivent être choisis judicieusement.
– Le code doit être propre et correctement indenté.
– Bonne gestion de la libération de la mémoire (utilisez valgrind pour vérifier que le nombre de `free` est égal au nombre de `malloc` : il faut d'abord compiler avec l'option `-g`, puis lancer valgrind avec en argument le nom du programme. Par exemple :

  ```
  gcc -g main.c
  valgrind ./a.out
  ```

  Valgrind peut également vous indiquer la ligne où un segmentation fault a eu lieu, pratique !
– Le projet doit évidemment être propre à chaque binôme. **Un détecteur automatique de plagiat sera utilisé**. Si du texte ou une petite portion de code a été empruntée (sur internet, chez un autre binôme), il faudra l'indiquer dans le rapport, ce qui n'empêchera pas l'application éventuelle d'une pénalité. Tout manque de sincérité sera lourdement sanctionné (conseil de discipline) – c'est déjà arrivé.

La documentation (rapports, commentaires...) compte pour une partie de la note finale.

Soutenance

Une soutenance d'une dizaine de minutes aura lieu pour chaque binôme, 3 binômes par 3 binômes dans l'ordre (3 jurys), à la date et à l'heure indiquées plus haut. Elle doit être préparée et menée par le binôme (*i.e.* fonctionnant parfaitement du premier coup, avoir préparé des jeux de tests intéressants, etc.). Nous aurons téléchargé vos projets (tels qu'envoyés sur MyCourse) sur les machines unix (donc vous n'avez pas besoin d'amener votre ordinateur et vous n'aurez pas besoin de vous loguer). Vous devrez en revanche compiler votre code.

Pendant la soutenance, ne perdez pas de temps à nous expliquer le sujet : nous le connaissons puisque nous l'avons écrit. Essayez de montrer ce qui fonctionne et de nous convaincre que vous avez fait du bon travail.

Bon courage