Université Jean Monnet

Faculté des Sciences et Techniques

# Projet de Programmation

Licence 1 Math-Info-Mass

2014-2015

---

Encadrants

Richard Baron, Romain Blin, Nohra Hage, Baptiste Jeudy, Michaël Perrot, Emilie Samuel, Geoffrey Tincani

---

Département d'Informatique

# Déroulement du projet

L'objet du projet est la réalisation d'un programme permettant de calculer et de visualiser des enveloppes convexes de points du plan. Vous devez le faire en binôme et dans les délais qui vous sont spécifiés.

## Binôme

Un binôme est composé de deux personnes. Une personne seule n'est donc pas un binôme. Trois ou quatre personnes ne forment pas un binôme non plus. Par contre, un redoublant est un binôme [1]. Toute dérogation à cette règle nécessitera l'accord (exceptionnel) de votre chargé de TP. Plus clairement, si vous ne la respectez pas, vous serez pénalisés lors de l'évaluation du projet. D'autre part, il est impératif que chaque membre du binôme travaille, dans la mesure où l'examen final (écrit) peut comporter des questions portant sur ce projet.

## Planning

Ce projet est relativement long. Il est donc impératif que vous travailliez dès le début, *pendant et en dehors* des heures de TP, chez vous ou à l'Université. Nombre d'étudiants n'ont pas suivi cette consigne les années précédentes ; ils ont fini par passer leurs jours et leurs nuits à écrire du CAML (souvent faux), pour une note finale pas très bonne, du fait d'une mauvaise organisation de leur travail.

## Organisation du projet

Ce projet est composé de deux parties. La première est une étude des tris de listes, en particulier de leur efficacité pratique, et la seconde, qui repose sur la première, vise le calcul des enveloppes convexes proprement dit. Chacune de ces parties est composée :

1. d'un certain nombre de connaissances théoriques en programmation et en algorithmique qu'il vous faut acquérir, de manière autonome, pour pouvoir travailler,
2. d'un ensemble minimal de fonctions qu'on vous demande de réaliser,
3. de plusieurs tests permettant de vérifier la correction et l'efficacité de vos fonctions,
4. d'un ensemble optionnel de fonctions qu'on vous *propose* pour compléter votre étude, si vous avez la curiosité et le temps nécessaire (donc après la réalisation du minimum exigé sur l'ensemble du projet).

## Evaluation du projet

Concernant l'évaluation du projet, elle sera établie sur la base d'un rapport écrit et d'une soutenance sur machine :

1. puisqu'un redoublant est un "re" qu'on double...

- Le rapport présentera l'ensemble des résultats de la première partie du projet (uniquement). Nous vous conseillons de l'écrire en LaTeX en utilisant vos cours d'Outils Logiciels et les sources du document que vous avez en main. Vous devrez le remettre à l'enseignant qui vous évaluera le jour de la soutenance oral.
- La soutenance concernera l'ensemble du projet, parties 1 et 2. Elle se déroulera en binôme, sur machine et pour une durée de 20 à 30 minutes. Dans un premier temps, vous devrez faire une démonstration de votre travail (ce qui exige une vraie préparation avant la soutenance). Dans un deuxième temps, vous devrez répondre aux questions que les encadrants ne manqueront pas de vous poser sur votre projet.

Reportez-vous à la fin de chaque partie du sujet pour plus de détails.

## Mise en œuvre

Sur un plan plus technique, tous les fichiers dont vous aurez besoin vous seront fournis par le Bureau Virtuel (Claroline) ; copiez-les dans votre répertoire de travail.

Vous êtes libres de l'architecture de machine que vous utiliserez (portable ou PC de salles de TP), mais vous devez vous assurer que votre travail fonctionne parfaitement sur l'architecture que vous choisirez au moment de la soutenance.

Concernant le système d'exploitation, Il vous est conseillé d'utiliser `Linux` (ou `Unix`) et non `Windows`.

Concernant, pour finir, votre propre répertoire de travail et vos fichiers, sachez qu'il est du plus mauvais effet, en soutenance, de montrer qu'on est archi-bordélique. Soyez rigoureux :

- faites du ménage dans votre répertoire, de temps en temps, en effaçant les fichiers inutiles (et seulement eux ...) ;
- faites des fichiers aérés, bien indentés, agréables à lire ;
- utilisez impérativement les noms des fonctions qui vous sont donnés dans le sujet, et choisissez des noms significatifs pour les fonctions autres.

## Rôle des encadrants

Notre rôle est double :

1. Nous règlerons les gros problèmes techniques liés à une erreur dans le sujet ou dans un des fichiers qu'on vous donne. Cela arrivera peut-être : nous vous prions de nous en excuser par avance.
2. Nous évaluerons votre travail à la fin du projet. Mais il n'y a pas de raison que cette évaluation ne se fasse que dans un sens : si vous avez des suggestions d'amélioration de ce projet, faites les nous parvenir pour que nous les utilisions l'an prochain.

Par contre, nous ne sommes pas là pour faire le projet à votre place. Cela signifie que nous *ne règlerons pas* vos problèmes de CAML, à moins que vous ne bloquiez dessus *depuis plusieurs jours*. Vous devrez vous débrouiller *seul* pour corriger vos fonctions : cela fait partie du travail d'un programmeur.

Un conseil, toutefois : si vous ne voulez pas avoir de problème, il faut *absolument* que vous testiez *rigoureusement* chacune de vos fonctions sur *de nombreux exemples*, afin d'être archi-sûr qu'elles fonctionnent correctement. Faites ce travail de débogage *chaque fois* que vous écrivez une nouvelle fonction et avant d'en écrire une nouvelle. Sachez qu'il n'y a rien de plus difficile à faire que de corriger une erreur en fin de projet, quand cette erreur vient d'une fonction fausse qui a été écrite au tout début du projet.

## Bon travail à tous !

# 1ère partie : sur les tris de listes

L'objectif de cette première partie du projet est double. Tout d'abord, elle consiste en une étude comparative des tris et de leur efficacité. Du succès de cette comparaison dépend celui de la seconde partie du projet. Ensuite, elle vise à écrire un nouveau module de traitement de listes.

## Les modules

Comme vous l'avez déjà vu en TP, CAML fournit des bibliothèques de fonctions prédéfinies, qu'on appelle des modules, comme par exemple `String` et `List`. L'existence de ces modules présentent de nombreux avantages. Par exemple, un programmeur peut utiliser ces fonctions quand il le désire, sans avoir à les (re-)programmer. De plus, l'implantation de ces fonctions est souvent d'une redoutable efficacité; un programmeur a donc tout intérêt à les utiliser pour augmenter l'efficacité de ses propres programmes.

Pour utiliser une fonction définie dans un module, il suffit de préfixer le nom de cette fonction par le nom du module qui la contient. C'est ce que vous faites quand vous utilisez les fonctions `String.length` et `String.sub`. Cette écriture signifie que les fonctions `length` et `sub` sont définies dans le module `String`. A noter qu'il existe aussi une fonction `length` dans le module `List` permettant de calculer la longueur d'une liste. Pour l'utiliser, il suffira donc d'écrire `List.length`.

On peut aussi *ouvrir* le module avec la commande `open`, ce qui permet d'éviter le préfixage. Mais il faut être prudent : un nom de fonction peut être défini dans plusieurs modules différents (comme `length`) sans désigner la même fonction pour autant. Cela peut donc introduire des confusions ...

```
# List.length [1; 2; 3] ;;
- : int = 3
# String.length "Je suis fan de caml" ;;
- : int = 19
# length [1; 2; 3] ;;
Unbound value length
# open List ;;
# length [1; 2; 3] ;;
- : int = 3
# open String ;;
# length [1; 2; 3] ;;
This expression has type 'a list but is here used with type string
```

### Définir ses propres modules ...

En dehors des modules prédéfinis, CAML permet aussi à un programmeur de *définir* des modules et de les utiliser. C'est le cas du module `Hasard` que nous vous donnons (voir Annexe A).

Un module se compose toujours de deux parties :

1. Une partie *spécification*, qui regroupe les déclarations des types et des fonctions fournies par le module (bibliothèque de programmes).

2. Une partie *implantation*, qui contient les programmes associés aux fonctions du module.

Ces deux parties se trouvent le plus souvent dans deux fichiers distincts dont les noms sont significatifs. Ainsi, la spécification du module `Hasard` se trouve (nécessairement) dans le fichier `hasard.mli` et son implantation, dans le fichier `hasard.ml` (voir Annexe A). Si vous regardez ces deux fichiers de plus près, vous constatez que l'implantation contient bien du code en CAML qui décrit chacune des fonctions déclarées dans la spécification. Ce fichier pourrait contenir d'autres fonctions non déclarées dans la spécification et qui seraient alors inutilisables en dehors du fichier. Cela permet de définir des fonctions auxilliaires qui sont invisibles à l'utilisateur du module.

Mais notez aussi que, sauf exception, vous n'êtes pas en mesure de comprendre parfaitement le code du fichier `hasard.ml`. C'est normal et ce n'est pas grave : si un module est bien conçu, que sa spécification est bien documentée, il n'est pas nécessaire qu'un programmeur sache comment les fonctions de ce module sont programmées pour les utiliser. Seule la spécification du module suffit. D'ailleurs, vous utilisez vous-mêmes des fonctions comme `String.length` et `String.sub` sans jamais vous être posé la question de savoir comment elles avaient été programmées.

### Compilation et chargement d'un module

En CAML, on utilise rarement des modules sans les avoir préalablement compilés. C'est le cas des modules prédéfinis `String` et `List`. Pour les autres modules, par contre, il faut effectuer cette opération de compilation "à la main", avec la commande `ocamlc -c`. Pour cela, on commence toujours par compiler la spécification du module, le `.mli`, ce qui génère un fichier `.cmi`. Il faut ensuite compiler son implantation, le `.ml`, ce qui génère un fichier `.cmo`. On peut ensuite utiliser le module dans l'interpréteur d'`ocaml`, en chargeant préalablement le fichier `.cmo` avec la commande[2] `#load`. Les modules prédéfinis sont, eux, automatiquement chargés lorsqu'on lance `ocaml`. On peut ensuite travailler :

```
> ocamlc -c hasard.mli
> ocamlc -c hasard.ml
> ls -l
total 4
-rw-r--r-- 1 baron profs 308 Mar 26 17:22 hasard.cmi
-rw-r--r-- 1 baron profs 761 Mar 26 17:22 hasard.cmo
-rw-r--r-- 1 baron profs 788 Mar 26 17:21 hasard.ml
-rw-r--r-- 1 baron profs 713 Mar 26 17:21 hasard.mli
> ocaml
Objective Caml version 3.04
# #load "hasard.cmo" ;;
# open Hasard ;;
# init_random () ;;
- : unit = ()
# random_list 100 10 ;;
- : int list = [44; 53; 58; 16; 31; 10; 47; 45; 80; 86]
```

## Travail demandé

Notre objectif est de réaliser un nouveau module `List_sup` contenant de nouvelles fonctions sur les listes. Nous écrirons la spécification complète de ce module plus tard. En attendant, ouvrez un nouveau fichier `list_sup.ml` dans lequel vous devez absolument sauvegarder toutes vos fonctions.

[2] La commande #use permet de charger des fichiers *non* compilés, donc des fichiers `.ml`

La principale fonction que va fournir le module `List_sup` est une fonction de tri de listes qui doit être la plus efficace possible. Cette fonction de tri sera utilisée dans la suite du projet sur des listes d'objets géométriques. Cependant, comme les ordres que vous devrez utiliser pour comparer deux objets géométriques ne sont pas encore définis, la plupart des fonctions suivantes seront paramétrées par une fonction de comparaison de type `'a -> 'a -> bool`.

Bref, dans la suite, nous allons implanter plusieurs fonctions de tri. Puis nous comparerons leur efficacité sur des exemples de manière à choisir la meilleure.

## Tri par sélection du minimum

1. Ecrire une fonction `selectionne : ('a -> 'a -> bool) -> 'a list -> 'a`.
   `(selectionne inf l)` calcule le minimum de la liste `l` par rapport à la fonction de comparaison `inf` (passée en paramètre). Par exemple, `(selectionne (<=) ['b';'a';'d';'c'])` retourne le caractère `'a'`, et `(selectionne (>) [2;1;4;3])` retourne l'entier 4.
2. Ecrire une fonction `supprime : 'a -> 'a list -> 'a list`.
   `(supprime x l)` retourne la liste `l` sans la première occurence de `x` quand elle existe, et ne fait rien sinon.
3. Ecrire une fonction
   `tri_selection_min : ('a -> 'a -> bool) -> 'a list -> 'a list`
   qui trie une liste par ordre croissant, conformément à la fonction de comparaison passée en paramètre, en sélectionnant le premier minimum, puis en sélectionnant le second minimum (c'est-à-dire le minimum de la liste obtenue en supprimant le premier minimum), puis le troisième minimum, et ainsi de suite.
4. Utiliser le module `Hasard` pour tester la fonction précédente sur des listes courtes, puis plus longues, des listes avec peu de doublons et des listes avec beaucoup de doublons.
5. Sauver (sans quitter) votre fichier `list_sup.ml`.

## Tri par partition-fusion

1. Ecrire une fonction `partitionne : 'a list -> ('a list * 'a list)` qui partage une liste en deux listes de tailles équivalentes. Pour cela, on met tous les éléments de rang pairs dans la première, et tous les éléments de rang impairs dans la seconde (et lorsqu'il n'y a qu'un seul élément, dans l'une ou l'autre).
2. Ecrire une fonction
   `fusionne : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list`
   qui prend deux listes triées et les fusionne en une seule liste, tout en préservant l'ordre.
3. Ecrire enfin la fonction
   `tri_partition_fusion : ('a -> 'a -> bool) -> 'a list -> 'a list`.
   `(tri_partition_fusion inf l)` partage `l` en deux listes de tailles équivalentes, trie chacune de ces deux listes récursivement, puis fusionne les deux résultats.
4. De même qu'avant, tester la fonction précédente sur des listes courtes. Vous pouvez ensuite la tester sur des listes plus longues, des listes avec peu de doublons et des listes avec beaucoup de doublons.
5. Sauver (sans quitter) votre fichier `list_sup.ml`.

## Choix d'une fonction de tri

Pour choisir la meilleure fonction de tri, vous devez tirer au hasard de grandes listes d'entiers, les trier par sélection du minimum et par partition-fusion, puis déterminer l'algorithme qui vous semble le meilleur en terme d'efficacité. Attention, vous ne ferez pas nécessairement les mêmes constats que votre voisin, et vous devrez montrer la pertinence de votre choix dans le rapport puis en soutenance. Aussi, soyez cohérents ...

Pour vous aider à faire ce choix, nous vous suggérons d'étudier le temps de calcul que nécessite chacun des algorithmes en fonction de la taille de la liste à trier. Pour faire ces mesures, utilisez le bout de code suivant ; il retourne le temps (en seconde) que met le processeur de l'ordinateur pour faire votre calcul :

```
# let temps_debut = Sys.time () in
  let _ = telle_fonction_de_tri ... in
  let temps_fin = Sys.time () in
  (temps_fin -. temps_debut) ;;
```

Une fois que vous avez choisi la meilleure fonction de tri, vous pouvez définir la fonction `tri : ('a -> 'a -> bool) -> 'a list -> 'a list` par un simple renommage (par exemple `let tri = tri_par_insertion ; ;`).

### Définition du module `List_sup`

Ce module doit contenir trois fonctions :
- la fonction `tri : ('a -> 'a -> bool) -> 'a list -> 'a list` précédente,
- une fonction `min_list : ('a -> 'a -> bool) -> 'a list -> 'a` qui retourne le minimum d'une liste et
- une fonction `suppr_doublons : 'a list -> 'a list` qui prend une liste *triée*, contenant d'éventuels doublons, et qui retourne cette liste sans ses doublons.

#### Questions

1. Après avoir relu attentivement l'introduction précédente sur les modules, écrire le fichier `list_sup.mli` contenant la spécification du module `List_sup`. Compiler ce fichier puis vérifier que le fichier `list_sup.cmi` a bien été créé.
2. Ajouter les fonctions `min_list` et `suppr_doublons` dans le fichier `list_sup.ml`. *Tester ces nouvelles fonctions.* Compiler le fichier `list_sup.ml` puis vérifier que le fichier `list_sup.cmo` a bien été créé.
3. Relancer l'interpréteur `ocaml`, charger les modules `Hasard` et `List_sup` (avec `#load`), puis vérifier une nouvelle fois que toutes les fonctions du module `List_sup` sont bien correctes.

## Pour aller plus loin ...

*Bien que facultatives*, nous encourageons toutes vos initiatives dans la poursuite éventuelle de ce travail : si elles sont originales et si vous ne perdez pas de vue l'essentiel (la réalisation du projet dans son ensemble), elles seront forcément récompensées.

Voici quelques pistes :
- Hormis les deux fonctions précédentes, vous êtes susceptibles de connaître deux ou trois autres fonctions de tri, celle par sélection du maximum, celle par insertion et celle par pivot (que nous rappelons ci-après). Une autre manière de poursuivre ce travail peut donc consister en l'implantation puis l'étude comparative de ces *cinq* fonctions de tri.
- Vous avez comparé les tris en observant les temps de calcul de votre machine sur des exemples. Or vous avez peut-être besoin de tracer des courbes pour présenter vos résultats, ce qu'on peut faire avec `gnuplot`. Il ne vous reste donc plus qu'à trouver sur INTERNET comment on utilise ce logiciel.
- D'autres fonctions de tri existent dans la nature, comme le tri à bulle. Pourquoi ne pas l'étudier, lui aussi, après l'avoir programmé ? Vous trouverez des informations à son sujet sur INTERNET.

**Rappel : Tri par pivot**

1. Ecrire une fonction `separe_inf_eq_sup` :
   `('a -> 'a -> bool) -> 'a -> 'a list -> ('a list * 'a list * 'a list)`.
   `(separe_inf_eq_sup comp x l)` retourne un triplet de listes `(i,m,f)` tel que `i` contienne tous les éléments de `l` qui sont strictement inférieurs à `x` (par rapport à la fonction de comparaison `comp` passée en paramètre), `m` contienne tous les éléments de `l` qui sont égaux à `x`, et `f` contienne tous les éléments de `l` qui sont strictement supérieurs à `x`.
2. Ecrire une fonction `tri_pivot : ('a -> 'a -> bool) -> 'a list -> 'a list`.
   `(tri_pivot infeq l)` sélectionne la tête `x` de `l` (qu'on appelle le pivot), calcule les listes d'éléments du reste de `l` qui sont strictement inférieurs, égaux et strictement supérieurs à `x` (selon la fonction de comparaison `infeq`), trie ces listes récursivement, puis les fusionne avec `x` pour retourner la liste `l` triée.

# Rapport

De quatre [ ] pages, il doit reprendre l'essentiel de cette première partie du projet (et uniquement de cette partie). Pour qu'il soit propre, nous vous conseillons de le rédiger en LaTeX. Pour vous aider, nous vous donnons les sources de ce sujet : il ne vous reste plus qu'à changer le contenu...

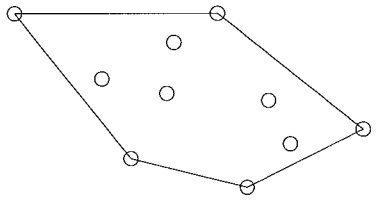
Quoiqu'il arrive, votre rapport ne contiendra aucun listing ni description linéaire des fonctions. Notez que vous pourrez l'utiliser comme support lors de votre soutenance. Dans ce rapport, vous devez, en fait, justifier les choix que vous avez faits, et en particulier celui de votre fonction de tri. En conséquence, vous présenterez vos différentes expérimentations et les chiffres, tableaux, courbes, etc., qui en découlent.

Vous pouvez aussi étudier les propriétés des listes qui favorisent ou au contraire qui pénalisent vos fonctions de tri. Par exemple, comment se comportent-elles quand les listes qu'on leur donne sont déjà triées par ordre croissant ou par ordre décroissant ? Comment se comportent-elles en face de listes avec énormément de doublons ou avec très peu de doublons ? Pourquoi, selon vous ?

N'omettez surtout pas vos propres idées : plus elles seront originales, plus elles seront récompensées.

# 2nde partie :
# Tracé d'enveloppes convexes

Les enveloppes convexes sont des objets géométriques qu'on rencontre fréquemment en informatique, par exemple, en recherche opérationnelle, en apprentissage automatique, en robotique, en statistiques inférentielles, etc. En voici la définition : soit $E$ un ensemble de points du plan contenant au moins trois points non alignés ; on appelle *envelopppe convexe* de $E$ le plus petit polygone convexe qui contient tous les points de $E$ (voir figure). Clairement, les sommets de l'enveloppe convexe de $E$ sont eux-mêmes des points de $E$.

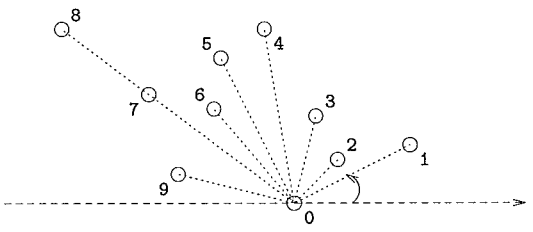


L'étude des enveloppes convexes est un sous-domaine de la *géométrie algorithmique.* De nombreux algorithmes de construction ont été proposés, et l'objectif de ce projet est d'en implanter un qui a été développé par Graham dans les années 60. Cet algorithme est très efficace : si l'ensemble de départ a $n$ points, le temps d'exécution de cet algorithme est de l'ordre de $n \log_2 n$. C'est même un algorithme optimal, car on peut démontrer qu'un algorithme de construction d'enveloppes convexes ne peut pas faire moins que cette borne.

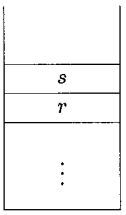
## L'algorithme de Graham

On considère une liste $L$ de points du plan sans doublon.

1. Soit $p_0 =$ `0` le point de plus petite ordonnée dans $L$, et de plus petite abscisse quand plusieurs points ont cette même plus petite ordonnée. On commence par trier les points de $L$ par angles croissants par rapport à l'axe horizontal passant par $p_0$.



2. On parcourt ensuite cette liste triée selon l'ordre précédent, en conservant dans une pile les points qui sont candidats à être les sommets de l'enveloppe convexe. Soyons plus précis, avant de traiter un exemple. Notons $s$ le sommet de la pile, et $r$, le sous-sommet de la pile.




   (a) Tout d'abord, on commence par empiler les points $p_0$ puis $p_1$.
   Initialement, $r$ vaut donc $p_0$ et $s$ vaut $p_1$.

   (b) Ensuite, pour chaque nouveau point $p$ considéré, trois situations peuvent se produire :

      i. si un marcheur allant de $r$ à $p$ en passant par $s$ tourne à droite en $s$, alors on depile le sommet de pile, et on recommence avec la nouvelle pile et la même liste de points à traiter.

      ii. si un marcheur allant de $r$ à $p$ en passant par $s$ tourne à gauche en $s$, alors on empile le point $p$, et on recommence avec la nouvelle pile et le reste des points à traiter.

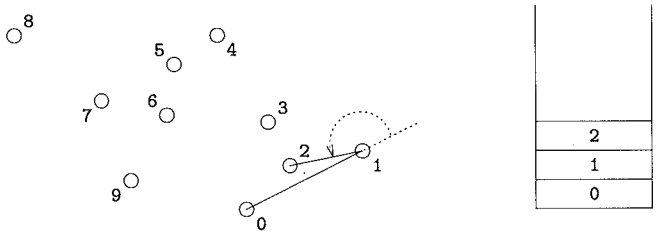
      iii. enfin, si un marcheur allant de $r$ à $p$ en passant par $s$ va tout droit en $s$, alors on dépile le sommet de pile, on empile le point $p$, et on recommence avec la nouvelle pile et le reste des points à traiter.

   (c) Quand il n'y a plus de point à traiter, la pile contient tous les sommets de l'enveloppe convexe dans l'ordre.

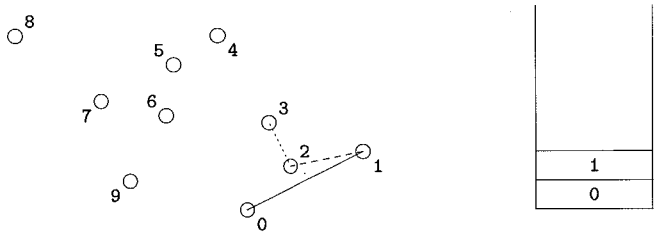
## Inutile de vous pendre ...

...nous allons détailler le fonctionnement de l'algorithme sur un exemple.

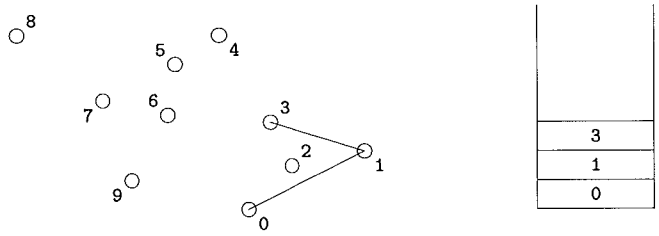
1. Comme nous l'avons dit, initialement, le sommet de la pile est le point 1 et le sous-sommet, le point 0. On considère le point 2. Un marcheur allant de 0 à 2 en passant par 1 tourne à gauche en 1. Donc on empile 2.



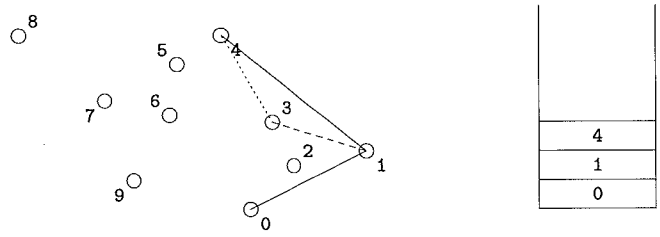
2. On considère maintenant le point 3. Le sommet de la pile est 2 et le sous-sommet 1. Clairement, un marcheur allant de 1 à 3 en passant par 2 tourne à droite en 2, donc on dépile 2.



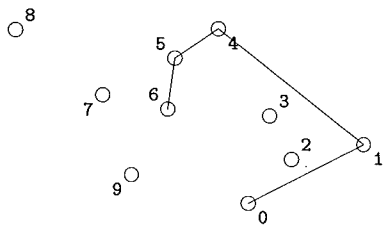

3. On considère de nouveau le point 3. Cette fois, le sommet de la pile est 1 et le sous-sommet 0. Un marcheur allant de 0 à 3 en passant par 1 tourne à gauche en 1 donc on empile 3.




4. On continue. On voit qu'en considérant 4, l'algorithme va dépiler 3 dans un premier temps, puis empiler 4 dans un deuxième temps.

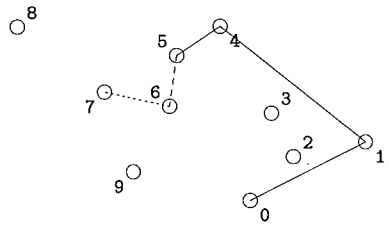



5. On considère maintenant le point 5. Le sommet de la pile est 4 et le sous-sommet 1. Un marcheur allant de 1 à 5 en passant par 4 tourne à gauche en 4 donc on empile 5. La pile contient donc désormais 5, 4, 1 et 0. Considérant le point 6, il est clair que l'algorithme empile 6.



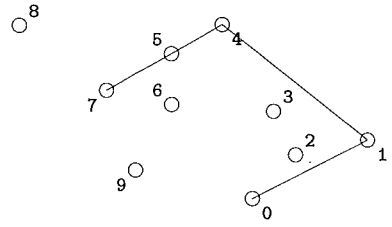

| |
|---|
| 6 |
| 5 |
| 4 |
| 1 |
| 0 |

6. On considère maintenant le point 7. Le sommet de la pile est 6, le sous-sommet 5, puis viennent 4, 1 et 0. Un marcheur allant de 5 à 7 en passant par 6 tourne à droite en 6 donc on dépile 6.



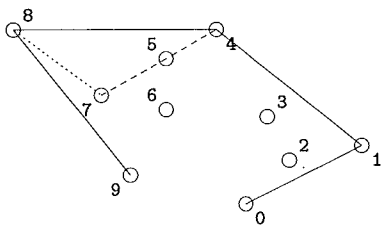

| |
|---|
| 5 |
| 4 |
| 1 |
| 0 |

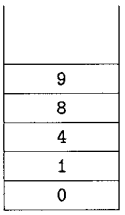
7. On considère de nouveau le point 7. La pile contient 5 puis 4, 1, 0. Un marcheur allant de 4 à 7 en passant par 5 va tout droit, donc on dépile 5 et on empile 7.




| |
|---|
| 7 |
| 4 |
| 1 |
| 0 |

8. On considère maintenant le point 8. A vous de vérifier que l'algorithme dépile 7, puis empile 8 et finit par empiler 9.




9. Comme `9` était le dernier point à traiter, l'algorithme s'arrête. La pile contient `9` puis `8`, `4`, `1`, `0`. Et vous constatez qu'effectivement, les sommets de l'enveloppe convexe sont bien `[0;1;4;8;9]`.

## Travail préparatoire

Comme nous l'avons dit, l'objectif de ce projet est d'implanter l'algorithme de Graham. Plusieurs modules sont nécessaires : `List_sup`, `Point` et `Pile`. Vous trouverez les spécifications des deux derniers dans l'Annexe B. Prenez le temps de les lire ...

L'intérêt du module `Point` est de définir les types de base qui vont vous permettre de travailler, à savoir, le type `point`, le type `nuage` (ensemble de points) et le type `polygone` (liste ordonnée des sommets d'un polygone). En outre, ce module offre des fonctions permettant de générer aléatoirement des nuages de points et de les dessiner sur une fenêtre graphique.

Malheureusement, ce module n'est pas utilisable tel quel dans l'interpréteur `ocaml` car on il utilise des fonctions graphiques qu'`ocaml` ne connaît pas par défaut. Aussi, pour pouvoir utiliser ce module, il faut créer un nouvel interpréteur, `mlgr`, ce qui se fait avec la commande `ocamlmktop`, comme dans l'exemple ci-dessous. Vous pourrez ensuite essayer les fonctions du module :

```
> ocamlmktop -o mlgr graphics.cma
> mlgr
Objective Caml version 3.04
# #load "point.cmo" ;;
# open Point ;;
# initialiser () ;;
- : unit = ()
# let l = gen_poisson 2000 ;;
val l : Point.nuage = [{abs = 3; ord = 52}; {abs = 1; ord = 5}; ...]
# tracer_nuage l ;;
- : unit = ()
# vider () ;;
- : unit = ()
```

Pour finir, ouvrez un fichier `proj.ml` qui contiendra toutes les fonctions du programme principal. Ecrire les lignes vous permettant de charger les modules `List_sup`, `Point` et `Pile`. Vous êtes maintenant prêts pour travailler.

## Le cœur du programme

Comme nous l'avons vu, plusieurs prérequis sont nécessaires pour que l'algorithme de Graham fonctionne : il faut calculer le point $p_0$, puis trier la liste des points par angles croissants par rapport à la droite horizontale passant par $p_0$, puis supprimer tous les doublons de la liste triée.

## Calcul de $p_0$

Par définition, $p_0$ est le point de plus petite ordonnée dans le nuage, et de plus petite abscisse quand plusieurs points ont cette même plus petite ordonnée. Comme vous disposez de la fonction `List_sup.min_list`, il suffit de définir l'ordre $\leq_{\texttt{coord}}$ qui induit cette notion de minimum.

Soient $p_1(x_1, y_1)$ et $p_2(x_2, y_2)$ deux points.
On pose $p_1 \leq_{\texttt{coord}} p_2$ ssi
- $y_1 < y_2$ ou alors
- $y_1 = y_2$ et $x_1 \leq x_2$.

### Question

Ecrire la fonction de comparaison `infc : point -> point -> bool` qui implante l'ordre $\leq_{\texttt{coord}}$.

## Tri des points par angles croissants

Connaissant $p_0$, il s'agit maintenant de trier les points par angles croissants par rapport à une droite horizontale passant par $p_0$. De même que dans la question précédente, vous disposez d'une fonction extrêmement efficace `List_sup.tri`. Il suffit donc de définir la fonction de comparaison utilisée par le tri, ce que nous faisons ci-dessous.

Soient $w$ un point de référence (par exemple, $w = p_0$) et $p_1$ et $p_2$ deux points.
On pose $p_1 \leq_w p_2$ ssi
- $p_1 = w$, ou
- $p_1 = p_2$, ou
- $p_1 \neq w$ et $p_2 \neq w$ et $p_1 \neq p_2$ et $\det(\overrightarrow{wp_1}, \overrightarrow{wp_2}) > 0$, ou enfin
- $p_1 \neq w$ et $p_2 \neq w$ et $p_1 \neq p_2$ et $\det(\overrightarrow{wp_1}, \overrightarrow{wp_2}) = 0$ et $\overrightarrow{p_1w}.\overrightarrow{p_1p_2} < 0$.

Dans cette définition, det et "." désigne respectivement le déterminant et le produit scalaire de deux vecteurs. Il n'est sans doute pas inutile de rappeler qu'étant donnés deux vecteurs $\overrightarrow{u}(x_1, y_1)$ et $\overrightarrow{v}(x_2, y_2)$,

$$\det(\overrightarrow{u}, \overrightarrow{v}) = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1y_2 - x_2y_1, \text{ et } \overrightarrow{u}.\overrightarrow{v} = x_1x_2 + y_1y_2$$

### Questions

1. Définir les fonctions `det` et `sca`, de type `point -> point -> point -> int`, de sorte que `(det q0 q1 q2)` et `(sca q0 q1 q2)` calculent respectivement le déterminant et le produit scalaire des vecteurs $\overrightarrow{\texttt{q0q1}}$ et $\overrightarrow{\texttt{q0q2}}$.
2. Ecrire ensuite la fonction de comparaison `infg : point -> point -> point -> bool`. `(infg w p1 p2)` retourne vrai ssi `p1` $\leq_{\texttt{w}}$ `p2`.
3. Déduire de tout ce qui précède la fonction `tri_points : point list -> point list`. Cette fonction prend une liste de points (avec doublons), calcule le point $p_0$, trie la liste par angles croissants par rapport à la droite horizontale passant par $p_0$, élimine tous les doublons de la liste (une fois qu'elle a été triée) et retourne la liste résultant de l'ensemble de tous ces traitements.

## Algorithme de Graham

Compte tenu des informations qui vous ont été données en première partie, seule la programmation de "un marcheur tourne à droite, à gauche ou va tout droit en un point $s$ quand il va de $r$ à $p$ en passant par $s$" peut poser problème. En fait, tout est déjà quasiment écrit :
- le marcheur tourne à droite en $s$ quand il va de $r$ à $p$ ssi $\det(\overrightarrow{sp}, \overrightarrow{sr}) < 0$,

- le marcheur tourne à gauche en $s$ quand il va de $r$ à $p$ ssi $\det(\overrightarrow{sp}, \overrightarrow{sr}) > 0$ et
- le marcheur va tout droit en $s$ quand il va de $r$ à $p$ ssi $\det(\overrightarrow{sp}, \overrightarrow{sr}) = 0$.

**Questions**

1. Ecrire la fonction `algo_graham : point list -> point pile -> point pile` qui implante l'algorithme de Graham.
2. En déduire la fonction `env_graham : point list -> point list` qui prend une liste de points (quelconque) et qui retourne la liste des sommets de son enveloppe convexe.

## Expérimentations

Utilisez les fonctions précédentes pour construire et visualiser des enveloppes convexes. Vous pouvez utilisez les fonctions de génération aléatoire de nuages du module `Point` ou en proposer de nouvelles. Par ailleurs, la fonction suivante vous simplifiera sans doute la vie :

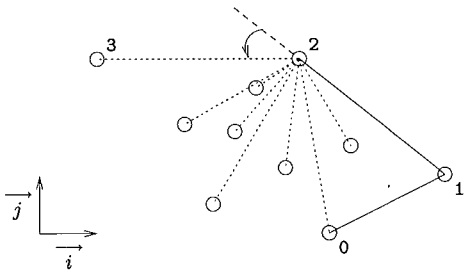
```
# let env g n =
    let l = (g n) in
    ( vider () ;
      tracer_nuage l ;
      tracer_polygone (env_graham l) ) ;;
val env : ('a -> nuage) -> 'a -> unit = <fun>
```

# Pour aller plus loin ...

Comme vous pourrez vous en rendre compte en surfant sur INTERNET, de nombreux autres algorithmes de construction d'enveloppes convexes ont été développés. Nous vous proposons ici d'en implanter un autre, celui de Jarvis. Mais nous vous rappelons que cette partie n'est pas obligatoire et que vos propres idées sont les bienvenues.

Pour expliciter le fonctionnement de l'algorithme de Jarvis, on considère une liste $L$ de points du plan, non triée et avec des doublons. L'algorithme de Jarvis parcourt les sommets de l'enveloppe convexe de proche en proche en les conservant dans une pile :

1. $p_0$ est le point de plus petite ordonnée dans $L$, et de plus petite abscisse quand plusieurs points ont la même plus petite ordonnée.
2. $p_1$ est le point de $L$ qui est différent de $p_0$ et tel que l'angle $(\overrightarrow{i}, \overrightarrow{p_0p_1})$ soit le plus petit possible.
3. $p_2$ est le point de $L$ qui est différent de $p_1$ et tel que l'angle $(\overrightarrow{p_0p_1}, \overrightarrow{p_1p_2})$ soit le plus petit possible.
4. $p_3$ est le point de $L$ qui est différent de $p_2$ et tel que l'angle $(\overrightarrow{p_1p_2}, \overrightarrow{p_2p_3})$ soit le plus petit possible (voir figure).




5. Et ainsi de suite ...jusqu'à ce qu'on rencontre de nouveau le point $p_0$.

**Questions**

1. Soit $s$ le dernier sommet trouvé de l'enveloppe convexe. Pour calculer le sommet suivant, il suffit de trouver le plus petit point du nuage selon une fonction de comparaison partiel $\preceq_s$ définie de la manière suivante. Soient $p_1$ et $p_2$ deux points du nuage. On pose $p_1 \preceq_s p_2$ si et seulement si
   - $p_2 = s$, ou
   - $p_2 = p_1$, ou
   - $p_2 \neq s$ et $p_1 \neq s$ et $p_2 \neq p_1$ et $\det(\overrightarrow{sp_1}, \overrightarrow{sp_2}) > 0$, ou enfin
   - $p_2 \neq s$ et $p_1 \neq s$ et $p_2 \neq p_1$ et $\det(\overrightarrow{sp_1}, \overrightarrow{sp_2}) = 0$ et $\overrightarrow{p_1 s}.\overrightarrow{p_1 p_2} > 0$.

   Ecrire la fonction `infj : point -> point -> point -> bool`.
   `(infj s p1 p2)` retourne `true` ssi `p1` $\preceq_s$ `p2`.
2. Ecrire la fonction `algo_jarvis : nuage -> point -> pile -> pile` qui retourne la pile de tous les sommets de l'enveloppe convexe. Cette fonction prend en argument :
   - le nuage de points,
   - le point $p_0$, calculé avant l'appel de `algo_jarvis`, et qui permet d'arrêter le calcul,
   - une pile, ne contenant initialement que $p_0$, et s'enrichissant, à chaque itération, d'un sommet supplémentaire de l'enveloppe convexe ; ce sommet sert de référence à l'itération suivante (c'est le $s$ de la question précédente).
3. Ecrire enfin la fonction `env_jarvis : nuage -> polygone` qui prend un nuage de points et qui retourne son enveloppe convexe.

# Soutenance

Elle se déroulera en binôme, sur machine, pendant 20 à 30 minutes. La moitié de ce temps sera consacrée à nos questions (et à vos réponses). Mais avant, vous devrez présenter l'ensemble de votre projet, sur la base du rapport d'une part, puis en montrant que vos fonctions permettent effectivement de construire des enveloppes convexes. *Préparez votre démonstration* car c'est vous qui mènerez la barque pendant cette première partie de soutenance. Essayez de mettre en valeur les atouts de votre projet. Montrez que vous avez eu des idées originales. Pour un examinateur, c'est ennuyeux de toujours voir la même chose...

# Annexe A

# Module Hasard

## A.1 Fichier `hasard.mli`

```
(* hasard.mli *)

      (*********************************)
      (*                               *)
      (* Spécification du module Hasard *)
      (*                               *)
      (*********************************)

(************************************************)

val init_random : unit -> unit

(* (init_random ()) permet d'initialiser
   le générateur aléatoire. On ne l'utilise
   qu'une fois en début d'une session d'ocaml. *)

(************************************************)

val random_list : int -> int -> int list

(* (random_list b n) retourne une liste de
   n entiers tirés au hasard entre 0 et (b - 1). *)

(************************************************)
```

## A.2 Fichier hasard.ml

```
(* hasard.ml *)

        (*********************************)
        (*                               *)
        (* Implantation du module Hasard *)
        (*                               *)
        (*********************************)

open Sys ;;

(************************************************)

let init_random () =
  let s = "/tmp/la_date_de_" ^ (getenv "LOGNAME") in
  let _ = command ("date +\"%M%H%j\" > " ^ s) in
  let c = open_in s in
  let n = int_of_string (input_line c) in
  ( close_in c ; remove s ; Random.init n ) ;;

(************************************************)

let rec random_list b n =
  if n <= 0
  then []
  else (Random.int b)::(random_list b (n - 1)) ;;

(************************************************)

let je_ne_suis_pas_déclarée_dans_la_spécification =
"donc on ne peut m'utiliser en dehors de ce fichier" ;;

(************************************************)
```

# Annexe B

# Modules Point et Pile

## B.1 Fichier `point.mli`

```
(* point.mli *)

              (*********************************)
              (*                               *)
              (* Spécification du module Point *)
              (*                               *)
              (*********************************)

(*****************************************************************)

type point = { x : int ; y : int }

type nuage = point list

type polygone = point list

(* un point est un enregistrement de coordonnées ; un nuage est
   un ensemble de points rangés dans une liste ; un polygône est
   représenté par la liste (ordonnée) de ses sommets consécutifs *)

(*****************************************************************)

val gen_rectangle   : int -> nuage
val gen_cercle      : int -> nuage
val gen_papillon    : int -> nuage
val gen_cerf_volant : int -> nuage
val gen_soleil      : int -> nuage
val gen_poisson     : int -> nuage

(* génèrent aléatoirement des nuages de points ; le paramètre
   des fonctions précédente est le nombre de points du nuage ;
   chacune d'elles produit un nuage d'une forme différente ;
   les nuages générés peuvent comporter des doublons *)

(*****************************************************************)
```

```
(* Fonctions graphiques : *)

val init     : unit -> unit
val vider    : unit -> unit
val terminer : unit -> unit

val tracer_point : point -> unit
(* dessine un point sur la fenêtre graphique *)

val tracer_nuage : nuage -> unit
(* dessine tous les points d'un nuage sur la fenêtre graphique *)

val tracer_polygone : polygone -> unit
(* trace un polygône sur la fenêtre graphique *)

(*****************************************************************)
```

## B.2 Fichier `pile.mli`

```
(* pile.mli *)

            (*******************************)
            (*                             *)
            (* Spécification du module Pile *)
            (*                             *)
            (*******************************)

type 'a pile

(* déclaration d'une pile :
   aucune indication sur l'implantion n'est donnée *)

(*********************************************************)

val vide : 'a pile

(* une constante du type pile *)

val empiler : 'a -> 'a pile -> 'a pile

(* ajoute un element au sommet de la pile *)

(*********************************************************)

exception Erreur_pile_vide

val depiler : 'a pile -> 'a pile

(* retourne la pile sans son sommet ; lève l'exception
   Erreur_pile_vide si on tente de dépiler une pile qui
   est vide *)

(*********************************************************)
```

```
val top    : 'a pile -> 'a
val subtop : 'a pile -> 'a

(* retourne les sommet et sous-sommet d'une pile sans toucher
   à la pile elle-même ; lève l'exception Erreur_pile_vide si
   on tente d'accéder au sommet d'une pile vide ou au
   sous-sommet d'une pile n'ayant qu'un élément *)

(*********************************************************)

val list_of_pile : 'a pile -> 'a list

(* retourne la liste de tous les éléments contenus dans une
   pile, de la base au sommet (ie, le dernier élément de la
   liste resultat est le sommet de la pile passé en argument *)

(*********************************************************)
```