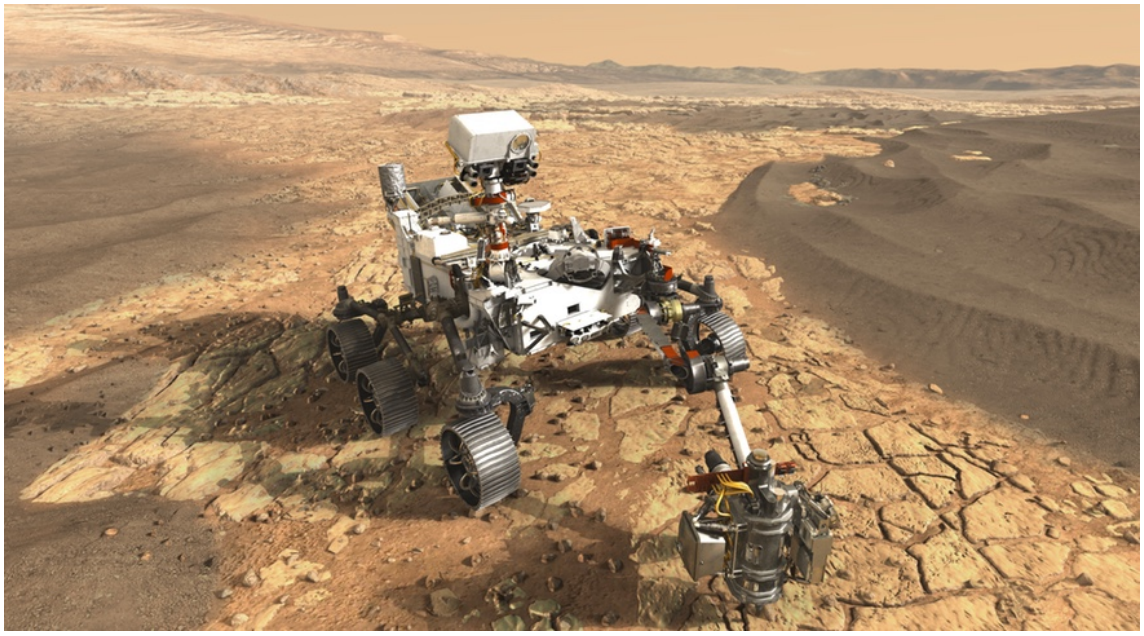
APP2 : Curiosity Reloaded

Florent Bouchez Tichadou

Octobre 2020



Source : `http://mars.nasa.gov`

Objectifs d'apprentissage : raisonnements algorithmiques sur listes chaînées, manipulations de séquences sous forme de listes chaînées, complexité algorithmique des manipulations de listes chaînées.

I APP2 : Curiosity Reloaded

I.1 Objectifs de la situation-problème

À l'issue de l'APP2, vous serez capable de :

- Expliquer la différence entre la représentation par tableau et celle par liste chaînée ;
- Donner et utiliser les algorithmes bas-niveau des opération de base sur les séquences représentées par listes chaînées (insertion, suppression, parcours...) ;
- Analyser les complexités d'algorithmes utilisant cette représentation ;
- Utiliser les informations de complexité pour choisir la structure de données bas-niveau appropriée pour stocker une séquence.

I.2 Organisation des séances

Cet APP comportera six séances encadrées (plus les cours de restructuration).

Entre chaque séance, du TRAvail Personnel (TRAP) est nécessaire et attendu !

- Séance Groupe d'ouverture (1h30) : découverte du problème et première analyse. Objectifs prévus : actes I et II.
- Séance Pratique en binômes (3h00) : découverte des listes chaînées, première implantation d'une séquence sous forme de liste chaînée.
- Séance Groupe de mise en commun (1h30) : réflexion commune sur la résolution des problèmes de l'acte III.
- Séance Pratique en binômes (3h00) : implantation des algorithmes choisis.
- Séance Groupe de mise en commun (1h30) : Terminez la réflexion sur la façon de résoudre tous les problèmes proposés. Préparez votre tableau pour votre rendu de groupe.
- Séance Pratique en binômes (3h00) : fin de l'implantation de votre programme.
- Séance(s) de clôture : évaluation (interrogation 30min) lors du prochain CM.

I.3 Ressources pour la recherche d'informations

- Polycopié du cours, chapitre sur les listes chaînées ;
- Polycopié du cours, chapitre sur la complexité (parties concernant les listes chaînées).

I.4 Rôles

Pour cette séquence APP, inscrivez les personnes volontaires pour chacun des quatre rôles (attention à ne pas reprendre un rôle que vous avez déjà effectué !) :

Modérateur _____

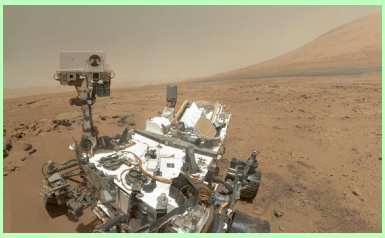
Gardien du temps _____

Scribe _____

Secrétaire _____

Situation — problème

« *Liquid water on Mars !* » Maintenant, on en est sûrs, il y a de l'eau liquide sur Mars. Petit problème : le plan d'exploration du robot Curiosity est complètement perturbé, il ne faudrait pas qu'il s'embourbe dans de la boue martienne ou tombe au fond d'une flaque par mégarde…



Les scientifiques de la NASA travaillent d'arrache-pied pour mettre à jour les programmes d'exploration de Curiosity. Basés sur les nouvelles données des cartes du printemps martien, ils ont complètement transformé les ordres du petit robot. Reste à envoyer à Curiosity la mise à jour. Malheureusement, le lien Mars–Terre est ténu, et les distances gigantesques font que les mises à jour échouent les une après les autres : elles sont trop grosses et arrivent corrompues à destination. Il va falloir trouver une autre solution.

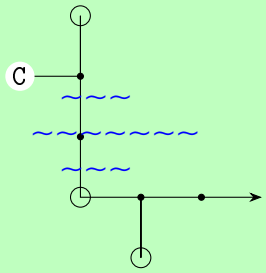
Tâches de Curiosity

Pour diminuer la taille des programmes, nous allons proposer à la NASA un langage spécifique très compact à base de chaînes de caractères.

Pour éviter toute confusion par la suite, nous appelerons les programmes de curiosity des **routines**. Ainsi, une *routine* de Curiosity est une séquence d'**ordres** basiques : tourner à **[G]**auche, à **[D]**roite, **[A]**vancer, effectuer une **[M]**esure, **[P]**oser une marque au sol (ou l'effacer).

On peut ainsi décrire les routines de Curiosity avec une séquence de lettres. Ci-dessous, un exemple de routine et à droite le parcours que devrait effectuer Curiosity : on suppose que Curiosity (initialement à la position `C`) est orienté vers l'est, les grands ronds représentent les points où le robot fait une mesure, les ~ représentent de l'eau liquide. Malheureusement, dans ce cas, Curiosity tombe à l'eau (plouf) avant même de réussir à faire sa deuxième mesure.

```
A G A M D D A A A G M A D A M G G A D A...
```



IMPORTANT : Dans le cadre de cet APP, on fera l'hypothèse que les routines à exécuter sont toujours *bien formées*, c'est-à-dire qu'elles respectent les règles de syntaxe définies dans ce document. En particulier, il n'est pas nécessaire/primordial de gérer le cas où une routine contient des caractères invalides.

Acte I

Dans cette première partie, le but est de simuler une routine simple de Curiosity afin de vérifier que tout va bien avant d'envoyer la mise à jour à Curiosity (imaginez la catastrophe si le robot fait une *syntax error* à des millions de kilomètres !).

Pour des raisons d'efficacité (voir actes suivants), Curiosity stocke ses routines sous forme de **listes**

chaînées.

Vous devez créer un simulateur basique permettant d'interpréter une routine de Curiosity comprenant uniquement des **A**, **G** et **D**.

Nous vous fournissons un environnement qui maintient une carte du lieu où se trouve Curiosity, qui définit trois fonctions en lien avec les actions du robot (`avance`, `gauche` et `droite`). À vous de gérer les aspects suivants :

— Chargement et conversion : à partir d'une chaîne de caractère initiale contenant l'intégralité de la routine, créer une représentation en mémoire sous forme de liste chaînée de cette routine. (Vous pouvez supposer qu'un programme vous est fourni qui lira dans un fichier une routine curiosity et appelera votre fonction avec en argument la chaîne de caractère qui correspond au contenu.)
— Exécution : construire un interpréteur capable d'exécuter la routine (complète) chargée dans le simulateur et convertie sous forme de séquence chaînée, en appelant dans le bon ordre les fonctions mentionnées ci-dessus.

Acte II

Pour empêcher Curiosity de tomber à l'eau, il va falloir doter notre langage de nouvelles instructions qui vont permettre à Curiosity de réagir en fonction de son environnement. Nous allons avoir besoin d'opérations permettant d'analyser et modifier l'environnement. Plus précisément, nous allons introduire deux nouvelles commandes : l'une pour effectuer des mesures du terrain à proximité de Curiosity et l'autre pour poser ou retirer une marque au sol.

Ces deux commandes diffèrent des précédentes car elles vont nécessiter des arguments, et, pour la mesure, produire un résultat. Nous allons avoir donc besoin d'une « mémoire » où chercher ces arguments et où stocker les résultats. Une manière très compacte de gérer une mémoire pour ce langage simple est d'utiliser une **pile** : les arguments sont récupérés en **dépilant**, et les résulats sauvegardés en **empilant**.

Pour introduire ce concept, dans cet acte nous nous contenterons de rajouter au langage de Curiosity des opérations arithmétiques simples : ce sont les commandes **+**, **−**, ***** (addition, soustraction, et multiplication). Nous allons également rajouter au langage les chiffres de **0** à **9**. Contrairement aux autres symboles qui sont des ordres directement exécutables, les chiffres sont non-exécutables et doivent être stockés sur la pile (**empilés**), en attendant d'être utilisés par des opérations. Une opération, quand à elle, va aller chercher sur le haut de la pile ses arguments. Par exemple une addition va **dépiler** les deux éléments en haut de la pile, les sommer, puis **ré-empiler** le résultat. [a]

Voyons sur quelques exemples comment cela fonctionne. (Note : pour décrire l'exécution d'une routine, on représente à chaque étape le reste des commandes à exécuter à droite, et à gauche l'état la pile. La pile est représentée horizontalement avec le « bas » de la pile à gauche et le « haut » de la pile à droite.)

Pile	Routine
vide	2 3 +
2	3 +
2 3	+
5	*vide*

Pile	Routine
vide	9 3 − 1 +
9	3 − 1 +
9 3	− 1 +
6	1 +
6 1	+
7	*vide*

Pile	Routine
vide	4 1 3 + *
4	1 3 + *
4 1	3 + *
4 1 3	+ *
4 4	*
16	*vide*

Pile	Routine
vide	2 3 * 3 4 + *
2 3	* 3 4 + *
6	3 4 + *
6 3 4	+ *
6 7	*
42	*vide*

a. Pour information, cette façon de représenter les calculs arithmétiques s'appelle la *notation polonaise inverse* et a

été longtemps utilisée sur des calculatrices HP notamment. Elle permet de faire de long calculs sans parenthèses et à moindre risque d'erreurs.

Acte III

Comme expliqué dans l'acte précédent, nous introduisons maintenant deux commandes : **P**ose et **M**esure. Ces deux nouvelles commandes diffèrent des premières car elles vont nécessiter des accès à une pile pour stocker le résultat, mais également pour y chercher les arguments.

À partir de maintenant, on utilisera pour introduire les commandes la notation [a b c **Z**] : ici, cela signifierait que la commande **Z** nécessite trois arguments a, b et c qu'elle va récupérer sur le dessus de la pile. L'addition vue à l'acte précédent serait ainsi notée [a b **+**].

Commande [f **P**] : pose ou enlève une marque au sol

Valeurs d'entrées pour f	
0	retire une marque
$\neq 0$	pose une marque

Ci-dessous, trois exemples. Les deux routines de gauche ont le même effet : Curiosity pose une marque, avance trois fois, puis retire une marque. Celle de droite pose une marque à chacun des quatre coins d'un carré.

Pile	Routine
vide	`1 P AAA 0 P`
`1`	`P AAA 0 P`
vide	`AAA 0 P`
vide	`0 P`
`0`	`P`
vide	*vide*

Variable	Routine
vide	`0 1 P AAA P`
`0`	`1 P AAA P`
`0 1`	`P AAA P`
`0`	`AAA P`
`0`	`P`
vide	*vide*

Variable	Routine
vide	`1 1 1 1 P AAAG P AAAG P AAAG P`
`1`	`1 1 1 P AAAG P AAAG P AAAG P`
`1 1 1 1`	`P AAAG P AAAG P AAAG P`
`1 1 1`	`AAAG P AAAG P AAAG P`
`1 1`	`AAAG P AAAG P`
`1`	`AAAG P`
vide	*vide*

Commande [d **M**] : effectue une « mesure » dans la direction d

Valeurs d'entrée pour d		Valeurs de sortie	
0	sur place	0	rien
1	devant	1	marque
2	devant droite		(posée par Curiosity)
3	droite	2	eau
4	derrière droite	3	rocher
5	derrière		
6	derrière gauche		
7	gauche		
8	devant gauche		

Exemple d'exécution sur la carte de la page 3 (eau à droite, rien à gauche).

Pile	Routine
vide	`A 3 M 7 M +`
vide	`3 M 7 M +`
`3`	`M 7 M +`
`2`	`7 M +`
`2 7`	`M +`
`2 0`	`+`
`2`	*vide*

Dans cet exemple, Curiosity avance d'un pas, fait une mesure à droite (eau), une mesure à gauche (rien), puis somme les deux résultats.

Dans ce troisième acte, vous devez modifier/étendre votre code de façon à gérer correctement les commandes **P** et **M**. Vous pouvez supposer que des fonctions `pose` et `mesure` vous sont fournies dans l'environnement de base.

Acte IV

Note : c'est le minimum attendu. Terminer cet acte vous permet d'avoir 12/20 au rendu de code.

Nous allons pouvoir maintenant utiliser les mesures faites par Curiosity pour éviter les poches d'eau (et les gros cailloux).

On introduit pour cela l'exécution conditionnelle **[?]** qui exécute un **groupe de commandes** (ou "bloc de code" - nous utiliserons les deux expressions de manière interchangeable). Plus précisément :

— Un groupe de commandes est délimité par des caractères accolades : **{** et **}**.
— Un groupe de commandes n'est pas exécutable et donc doit être placé sur la pile.
— La commande **?** est exécutable, et récupère sur le haut de la pile trois arguments : b, V et F. b doit être un entier et V et F des groupes de commandes. Si b est nul, alors F doit être exécuté, sinon, c'est V qui est exécuté.

Commande [n V F **?**] : exécute V si $n \neq 0$, et F si $n = 0$.

L'exemple ci-dessous présente une routine capable d'éviter un obstacle : s'il n'y a rien devant Curiosity (exécution de gauche), il avance deux fois, sinon il contourne l'obstacle par la gauche (exécution de droite, avec de l'eau).

Pile	Routine
vide	`1M {GADAADAG} {AA} ?`
`1`	`M {GADAADAG} {AA} ?`
`0`	`{GADAADAG} {AA} ?`
`0 {GADAADAG}`	`{AA} ?`
`0 {GADAADAG} {AA}`	`?`
vide	`AA`
vide	*vide*

Pile	Routine
vide	`1M {GADAADAG} {AA} ?`
`1`	`M {GADAADAG} {AA} ?`
`2`	`{GADAADAG} {AA} ?`
`2 {GADAADAG}`	`{AA} ?`
`2 {GADAADAG} {AA}`	`?`
vide	`GADAADAG`
vide	*vide*

Dans ce quatrième acte, vous devez gérer l'ajout de cette nouvelle fonctionnalité.

Acte V

Pour obtenir un langage complet, il nous reste quelques commandes à introduire. Le principe de chaque commande est détaillé ci-dessous. Implantez toutes ces nouvelles commandes, ce qui vous permettra de passer tous les tests fournis (et, sait-on jamais, de tomber sur des suprises...)

Commande [a b **X**] : échange les deux éléments au sommet de la pile

Commande [e **!**] : extrait l'élément au sommet de la pile et l'exécute

Pile	Routine
vide	`{ G } { D } X!!`
`{G}`	`{ D } X!!`
`{G} {D}`	`X!!`
`{D} {G}`	`!!`
`{D}`	`G !`
`{D}`	`!`
vide	`D`
vide	*vide*

Pour faire une « boucle » :

Commande [*cmd* n **B**] : exécute *cmd*, décrémente n sans enlever `B` de la routine si $n > 0$. Sinon enlève *cmd* et n de la pile, et `B` de la routine.

Pile	Routine
vide	`{GAD} 5 B`
`{GAD} 5`	`B`
`{GAD} 4`	`GAD B`
`{GAD} 4`	`B`
`{GAD} 3`	`GAD B`
...	
`{GAD} 0`	`B`
vide	*vide*

Commande [$a_n \dots a_2 a_1$ n x **R**] :
effectue une « rotation » de x pas vers la gauche des n éléments en haut de la pile : le haut de la pile devient $a_{n-x} \dots a_1 a_n a_{n-1} \dots a_{n-x+1}$

Pile	Routine
vide	`9 4 {D} {A} 3 1 R`
`9 4 {D} {A}`	`3 1 R`
`9 4 {D} {A} 3 1`	`R`
`9 {D} {A} 4`	*vide*

Commande [e **C**] : clone e sur la pile

Pile	Routine
vide	`{GAD} 5 C 31R C`
`{GAD} 5`	`C 31R C`
`{GAD} 5 5`	`31R C`
`{GAD} 5 5 3 1`	`R C`
`5 5 {GAD}`	`C`
`5 5 {GAD} {GAD}`	*vide*

Commande [e **I**] : ignore l'élément en haut de la pile.

Pile	Routine
vide	`{GAD} I 8 I`
`{GAD}`	`I 8 I`
vide	`8 I`
`8`	`I`
vide	*vide*

Complément d'informations

Vous êtes libres de vous organisez comme vous le souhaitez. Il est important que vous définissiez dans votre groupe un **planning de travail**, pour savoir le temps personnel que vous passerez à lire le cours, réfléchir au problème, etc. En particulier, il est primordial de bien comprendre comment fonctionnent les listes chaînées pour faire cet APP.

Vous trouverez pour les TPs dans le répertoire `/Public/301_INF_Public/APPs/APP2` (ou sur Caseine en `.zip`) plusieurs fichiers qui vous seront utiles :

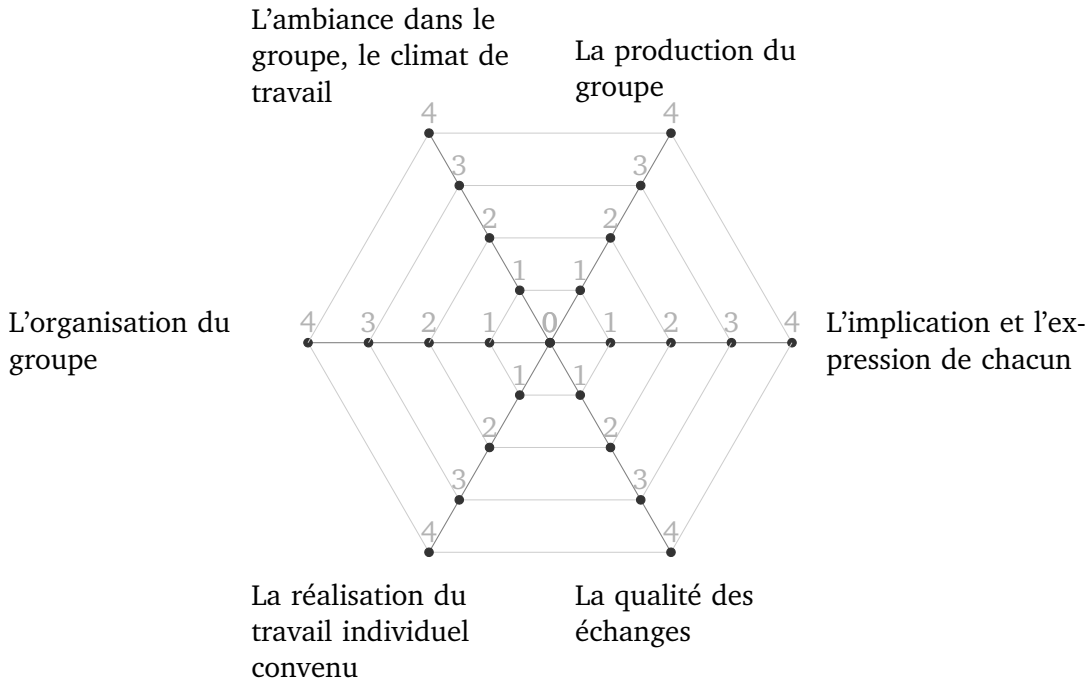
- `main.c/.py` la routine principale à qui l'on passe un fichier de test en argument (e.g., `./main(.py) test/simple.test`)
- `listes.{c/h/py}` un module basique de gestion de listes chaînées ; **vous pouvez le modifier** pour ajouter des fonctions aux listes chaînées ou des nouvelles structures de données ;
- `interprete.{c/h/py}` c'est là que les routines de Curiosity sont interprétés ; **vous pouvez le modifier**, c'est le fichier principal pour vous ;
- `curiosity.{c/h/py}` un module de gestion du robot Curiosity ; vous n'avez normalement pas besoin de le modifier mais vous devez utiliser ses fonctions dans votre interpréteur ;
- `Makefile` qui gère les dépendances de compilation pour le C (pas utilisé pour Python) ;
- `tests/` un répertoire contenant des fichiers de tests pour Curiosity. Le fichier `tests/LISEZMOI.txt` liste les fichiers de tests dans l'ordre croissant de difficulté.
- `tests/performance/` un répertoire contenant des fichiers de grande longueur pour réaliser des tests de performance.

Soyez attentifs lors de votre travail personnel (lecture du cours et réflexion sur le problème) aux difficultés que vous pouvez avoir. Notez-les sous forme de questions que vous pouvez soit poser sur Perusall soit nous envoyer par email pour les traiter pendant les cours de restructuration.

II Auto-évaluation du travail de groupe — Circept

II.1 Faire un bilan du travail de groupe

— **Individuellement :** remplissez le circept (cf. informations complémentaires page suivante) et répondez aux questions sur le travail de groupe (10 min.) ;
— Comparez vos réponses et tirer un bilan du travail de groupe ;
— Faites part de votre analyse au tuteur.



II.2 Donner deux points positifs de votre travail de groupe :

—
—

II.3 Donner deux points négatifs de votre travail de groupe :

—
—

II.4 Quels engagements prendriez-vous pour améliorer votre travail de groupe ?

—
—

III Exercices d'entraînement

Le chapitre sur les listes chaînées du poly de cours comporte de nombreux exercices. Nous vous recommandons de faire au moins quelques exercices autour de chaque notion importante (voir section suivante) pour vous entraîner.

IV Vos apprentissages

Pour chaque objectif de cet APP, estimez vous-même le niveau de vos apprentissages après cette séquence.

Au terme de l'APP, vous êtes capable de :	*Oui*	*Non, mais voici ce que je vais faire pour y remédier*
1. Expliquer la différence entre ensembles, séquences, tableaux et listes chaînées ;	☐	_____
2. Dessiner sur un schéma les effets d'un algorithme utilisant des listes chaînées ;	☐	_____
3. Expliquer la différence entre une référence vers une cellule et une cellule ;	☐	_____
4. Écrire un algorithme pour résoudre un problème simple qui manipule des listes chaînées ;	☐	_____
5. Vérifier qu'un algorithme sur listes chaînée respecte bien les cas limites (liste vide, changement à la tête, changement en queue, etc.) ;	☐	_____
6. Changer l'ordre des éléments d'une séquence sous forme de liste chaînée par modification des liens de chaînage ;	☐	_____
7. Donner les schémas de recherche, d'ajout, de suppression, etc. dans une séquence sous forme de liste chaînée ;	☐	_____
8. Donner les complexités des fonctions de base de travail sur les séquences avec la représentation par listes chaînées.	☐	_____