DS 2

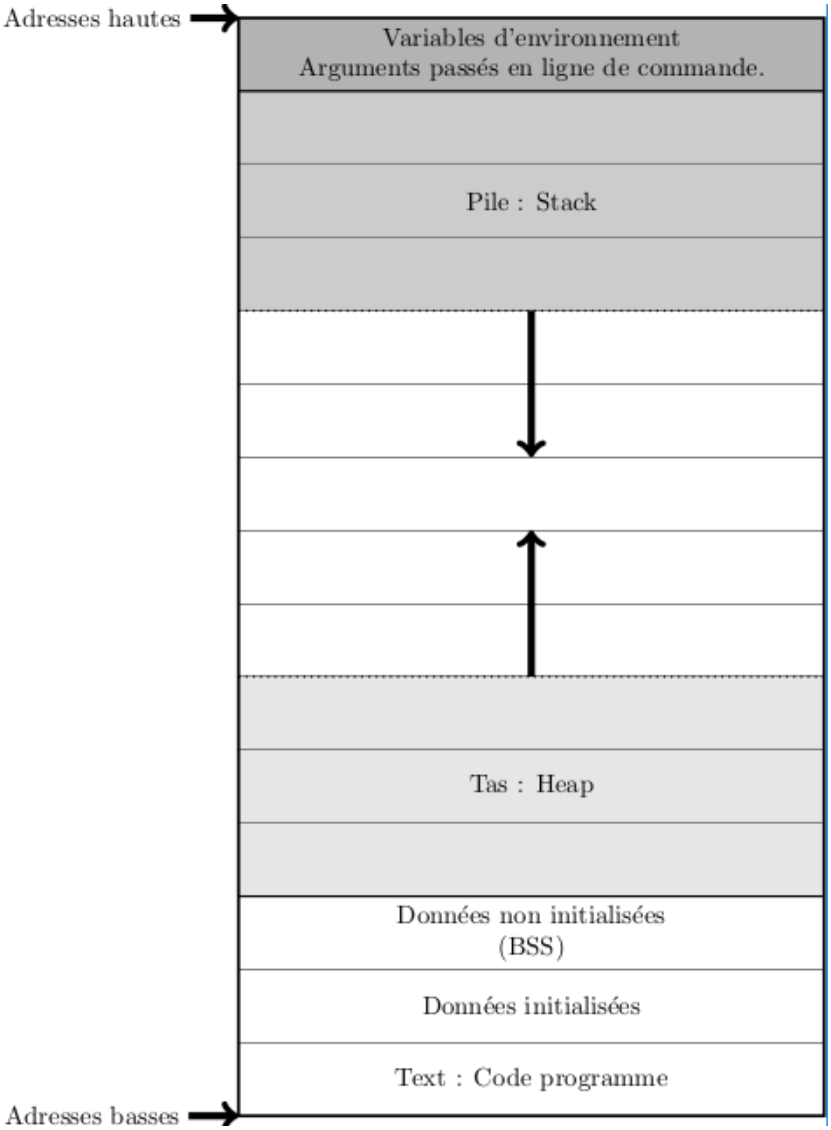
Informatique MP2I

Julien Reichert

Toutes les questions de programmation sont à résoudre dans le langage précisé dans la question ou dans le titre de la section.

Questions de cours

On rappelle l'organisation de la mémoire :



Question de cours A1 : On considère le programme en C ci-après. Pour chaque balisage `(X)` en commentaire, préciser dans quelle zone de la mémoire la valeur définie à la ligne en question est stockée.

```
#include <stdio.h>
#include <stdlib.h>

int gi = 42; // (1)
int *gn; // (2)

void f()
{
  int taille = gi / 7; // (3)
  int* tab = malloc(taille * sizeof(int)); // (4) (pointeur) et (5) contenu du tableau
  for (int i = 0 ; i < taille ; i += 1)
  {
    tab[i] = i*i;
  }
  gn = tab;
}

int main()
{
  f();
  printf("Adresse du tableau : %p.\n", gn);
  printf("La preuve, à la position 5 il devrait y avoir 25 et il y a %d.\n", gn[5]);
  free(gn);
  return 0;
}
```

Question de cours A2 : Dans le programme précédent, préciser la portée de `gn` et de `taille`.

Question de cours A3 : Expliquer le rôle de `>`, et la différence avec `»`, **dans une commande écrite au niveau d'un terminal Unix**. Trois points en moins directement si la réponse écrite sur la copie est qu'on a affaire à des opérateurs de comparaison.

Questions de cours en OCaml

Question de cours B1 : Si `s` est une chaîne de caractères et que `i` est un entier entre `0` inclus et la taille de `s` exclue, comment accède-t-on au caractère d'indice `i` de `s` ?

Question de cours B2 : Comment sépare-t-on deux « instructions » (en admettant ce terme abusif) ?

Question de cours B3 : Expliquer précisément (éventuellement exemple à l'appui) comment on balise un morceau de programme dans lequel d'éventuelles exceptions peuvent être rattrapées.

Questions de cours en C

Question de cours C1 : Que retourne la fonction `malloc` dans le cas, certes improbable, où l'allocation échoue ?

Question de cours C2 : Soit `s` une chaîne de caractères de longueur `n` au sens de la fonction `strlen`. Que vaut `s[n]` ?

Question de cours C3 : Que se passerait-il si on écrivait `int x = {1, 3, 5}[2];` et pourquoi ?

Exercices

Exercice $\alpha 1$: Prouver que disposer d'une expression qui s'évalue en un entier naturel qui décroît strictement aux tours de boucle impairs et reste identique aux tours de boucle pairs permet de garantir la terminaison d'une boucle conditionnelle.

Exercice $\alpha 2$: Sans justification, donner la complexité d'une fonction récursive lorsqu'elle est donnée par la formule de récurrence $c_n = c_{\frac{n}{2}} + \Theta(n)$, et décrire un algorithme qui aurait cette complexité, même s'il est stupide. Inutile de l'implémenter en C ni en OCaml, l'idée générale suffit.

Exercice $\alpha 3$: La copie en cours de rédaction ne m'appartient pas, mais j'aimerais bien la consulter tout de même. Écrire une commande qui me le permet (bien entendu, je suis l'administrateur).

Oui, c'est flou, mais cela donne d'autant plus de liberté dans la syntaxe. Bien entendu, on se base sur Unix.

Exercices en OCaml

Exercice $\beta 1$: On rappelle que `List.fold_left` prend en argument une fonction `f` de signature `'a -> 'b -> 'a`, une valeur initiale `x` de type `'a` et une liste `l` d'éléments tous de type `'b` (dont on note `a1` la tête et `an` l'élément au fond), et retourne le résultat de l'application imbriquée de `f` sur l'initialisateur et les éléments de la liste, selon la formule `f(... f( f(x a1) a2) ...) an)`. Écrire une fonction de même effet, bien entendu sans se servir d'une fonction similaire.

Exercice $\beta 2$: On rappelle qu'un type `'a option` consiste en deux constructeurs : `Some` paramétré par un élément de type `'a`, et `None` sans paramètre, de sorte de pouvoir avoir « quelque chose ou rien », naturellement sans compatibilité avec les autres types. Écrire une fonction prenant en argument un tableau ainsi qu'une valeur du type commun des éléments du tableau et calculant la position médiane des occurrences de la valeur dans le tableau, en renvoyant `None` en cas d'absence. La position médiane est celle du milieu parmi toutes les positions, de gauche si le nombre d'occurrences est pair.

Il existe une version élégante de complexité en espace constante et n'étudiant chaque élément du tableau qu'une fois, mais elle n'est pas attendue. Le seul objectif est d'avoir une complexité linéaire en temps.

Exercice $\beta 3$: Écrire une fonction prenant en argument un tableau `t`, dont (1) les éléments proviennent d'un ensemble totalement ordonné et (2) on suppose qu'il est croissant, et une valeur `x` du type des éléments du tableau et qui retourne le couple formé par le premier et le dernier indice où `x` figure (si `x` n'est pas dans `t`, on pourra au choix retourner le couple `(-1, -1)` ou déclencher une erreur). Si la complexité n'est pas logarithmique en la taille de `t`, la réponse ne sera pas prise en compte.

Exercices en C

Exercice $\gamma 1$: Écrire une fonction qui prend en argument deux tableaux d'entiers et leur taille, supposée commune (inutile de le vérifier), et qui échange leur contenu élément par élément. Attention, il ne s'agit pas de les rediriger !

Exercice $\gamma 2$: Écrire une fonction qui prend en argument un tableau d'entiers et sa taille (supposée d'au moins deux, inutile de le vérifier) et qui retourne l'écart relatif entre deux éléments consécutifs (celui de droite moins celui de gauche) qui est le plus grand en valeur absolue. En cas d'égalité, on prendra l'écart intervenant le plus tôt dans le tableau.

Exercice $\gamma 3$: Écrire une fonction qui prend en argument un tableau de chaînes de caractères et sa taille (supposée d'au moins un, inutile de le vérifier) et qui retourne l'indice au sein du tableau de la plus grande chaîne de caractères dans l'ordre lexicographique, la première rencontrée en cas d'égalité.

On pourra considérer que la gestion du tableau qui est passé en argument a été bien faite, sans avoir à s'en soucier…

Problème : Autour de générateurs pseudo-aléatoires

Dans ce problème, diverses façons d'engendrer des nombres en simulant le hasard seront abordées de manière indépendante.

Commençons par les suites arithmético-géométriques (on parle de générateur congruentiel linéaire). Il s'agit de considérer une suite arithmético-géométrique définie par un entier naturel u_0 (la graine) arbitraire et pour tout $k \in \mathbb{N}$, $u_{k+1} = au_k + b$, avec là aussi a (le multiplicateur) et b (l'incrément) deux constantes entières positives arbitraires. La suite pseudo-aléatoire voit toutes ses valeurs prises modulo un entier naturel n choisi. [1]

Question P1 : Justifier l'intérêt de prendre pour a, b et n des nombres premiers.

Question P2 : Justifier l'intérêt de prendre éventuellement pour n une puissance de deux.

Question P3 : On note que le calcul du reste modulo n peut se faire à chaque étape et non pas sur u_k une fois qu'il est obtenu. Justifier que le résultat final est le même. En comparant les avantages et inconvénients par rapport à une seule division euclidienne, quelle est la méthode la plus pertinente ?

Question P4 : Écrire en OCaml une fonction `gcl` prenant pour arguments les valeurs `u0`, `a`, `b`, `n` et un entier naturel `t` et retournant un tableau contenant `t` valeurs qui sont les premières engendrées par le générateur congruentiel linéaire associé aux arguments à partir de u_1.

La méthode de Von Neumann consiste à prendre pour u_0 un nombre tenant sur un certain nombre de chiffres (en base 10), noté n et nécessairement pair, à l'élever au carré pour obtenir un nombre sur $2n$ chiffres (quitte à ajouter des zéros au début), et à extraire les n chiffres du milieu pour donner le nombre suivant, qui subira la répétition. [2]

Question P5 : La méthode originale utilisait $n = 10$. Quel problème se pose en faisant le calcul sur un type entier en C ou en OCaml ?

Question P6 : Écrire en C une fonction `vng` prenant pour arguments la valeur `u0` et un entier naturel `t` et retournant un tableau contenant `t` valeurs qui sont les premières engendrées par le générateur de Von Neumann associé à $n = 8$, à partir de u_1. On choisira un type adapté !

L'efficacité des générateurs pseudo-aléatoires se mesure par des tests statistiques dont le plus connu est le test (en pratique l'ensemble de tests) Diehard.
Pour se donner une idée du fonctionnement d'un test statistique, nous allons implémenter les plus simples.

Question P7 : Écrire en C une fonction `test1` prenant en argument un tableau d'entiers `tab`, sa taille `l` et le nombre `n` de valeurs possibles qui sont nécessairement les premiers entiers naturels, et qui retourne l'histogramme des valeurs prises. La fonction permettra, par sa valeur de retour, d'avoir accès à un tableau contenant à tout indice `i` entre zéro inclus et `n` exclu le nombre d'éléments de `tab` égaux à `i`. On supposera `l` inférieure à un milliard.

Question P8 : Écrire en OCaml une fonction `test2` prenant en argument un tableau d'entiers `tab` et le nombre `n` de valeurs possibles qui sont nécessairement les premiers entiers naturels, et qui retourne l'histogramme des couples de valeurs prises consécutivement. Le rendu final sera par exemple une liste dont les éléments seront des couples dont la première composante sera elle-même un couple `(i, j)` pour tout `i` et tout `j` entre zéro inclus et `n` exclu, et la deuxième composante sera le nombre de fois où la valeur `i` aura été suivie de la valeur `j` dans `tab`.

Dans un contexte similaire, un fun fact pour finir : une séquence pseudo-aléatoire a souvent plus de bruit qu'une séquence engendrée par un humain pour faire croire à des valeurs aléatoires, où l'envie de respecter une régularité des valeurs se voit un peu trop. [3]

1. Attention : les fonctions renvoyant un entier entre 0 et un certain seuil n'utilisent pas forcément ce seuil pour valeur de n, mais procèdent plutôt au calcul du reste dans une nouvelle division euclidienne.
2. Pour des générateurs plus performants, on pourra se renseigner sur internet au sujet du Mersenne Twister.
3. Par ailleurs, des données numériques rencontrées dans la vraie vie voient souvent leurs valeurs suivre la loi dite de Benford, selon laquelle le chiffre le plus significatif a tendance à être plus souvent 1 que 9, selon une distribution clairement décroissante de fréquence. Cette loi a apparemment été utilisée pour tenter de détecter des fraudes.