ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation Orientée Système, sections IN et SC
Chappelier J.-C.

Programmation Orientée Système (IN/SC) :

CORRIGÉ DE LA SÉRIE NOTÉE

11 avril 2022

SUJET B

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre série annulée dans le cas contraire.

1. Vous disposez de 1h45 pour faire cette série notée (8h15 - 10h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.
   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; utilisez aussi le verso des feuilles, **MAIS** n'utilisez *que* le verso de la feuille sur laquelle se trouve la question, et non **pas** celui de la feuille précédente !
   Ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
   Si nécessaire, il y a des pages supplémentaires en fin de copie.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. Cette série notée ne comporte qu'un seul exercice en six questions.

1 – Recherche itérative de maxima [sur 70 points]

Les algorithmes génétiques sont une technique d'optimisation de fonctions à valeurs réelles, par recherche pseudo-aléatoire. On cherche ici à écrire des *parties* d'un programme d'optimisation par algorithmes génétiques.

Pour simplifier nous supposerons ici vouloir maximiser une fonction de $\mathbb{R}^3$ dans $\mathbb{R}$, accessible au travers d'un pointeur sur fonction, `selection`, global à tout le programme.

Pour trouver un maximum de cette fonction, un algorithme génétique utilise un ensemble (`Sample`) de « candidats » (`Candidate`) qui, dans notre cas simple, sont des points de $\mathbb{R}^3$ (trois `double`, donc).

Un algorithme génétique « mélange » alors itérativement un tel ensemble `Sample` pour en créer un nouveau, dont il ne garde à chaque étape que les meilleurs candidats.

Nous ne vous demandons dans cet exercice que de ne coder que *certaines parties spécifiques* d'un tel programme, lesquelles seront détaillées dans la suite. Un exemple de `main()` possible est donné en toute dernière question (page 7) ; vous pouvez bien sûr vous y référer si cela vous aide.

1.1 – Types de données [sur 12 points]

Avant tout, il faut définir les types de données utilisés.

[1.5 points] Définissez ci-dessous à gauche le type `Candidate` comme simplement trois `double`.
[2.5 points] Définissez ensuite (à droite) le type `Sample` comme un tableau dynamique de `Candidates`.

```
typedef struct {
    double x;
    double y;
    double z;
} Candidate;
```

```
typedef struct {
    size_t     size;
    Candidate* content;
} Sample;
```

[5 points] Définissez ensuite une fonction `Sample zero_sample(size_t nb);` qui crée un ensemble de candidats de taille donnée en paramètre, en initialisant tous les candidats à trois `0.0`[1] :

```
Sample zero_sample(size_t nb)
{
    Sample p = { 0, NULL };
    p.content = calloc(nb, sizeof(Candidate));
    if (p.content != NULL) p.size = nb;
    return p;
}
```

[3 points] Définissez enfin une fonction `delete()` qui prend un `Sample` en paramètre et libère son contenu. Elle ne retourne rien.

```
void delete (Sample* p)
{
    free(p->content);
    p->content = NULL;
    p->size = 0;
}
```

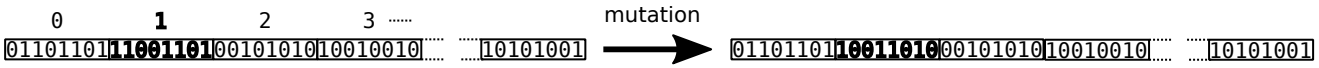
1. Si jamais, la représentation de `0.0` correspond à toute la mémoire à `0`.

1.2 – Mutation d'un candidat [sur 4 points]

Les algorithmes génétiques utilisent deux opérations pour générer de nouveaux candidats : le croisement et la mutation. Nous nous intéressons ici à cette dernière. La question suivante portera sur le croisement.

Pour muter un `Candidate`, nous considéreons celui-ci simplement comme une suite d'octets dont nous allons, pour simplifier, ne modifier qu'un seul octet.

Définissez ci-dessous la fonction `mutate_candidate()` qui retourne un nouveau `Candidate`, modification d'un `Candidate` reçu en paramètre (lequel est inchangé) tel que son second **octet** soit mutiplié par 2 :



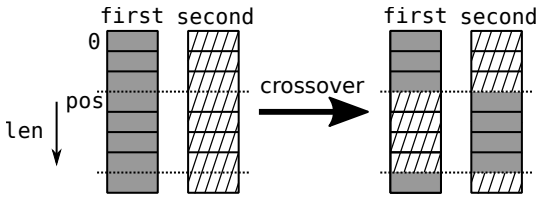

```
Candidate mutate_candidate(const Candidate* a)
{
    Candidate new = *a;

    char* const indiv  = (char*) &new; // view Candidate as an array of char

    // a very simple mutation
    indiv[1] *= 2; // or <<= 1;

    return new;
}
```

1.3 – Croisement candidats [sur 15 points]

Nous nous intéressons maintenant aux croisements entre candidats ; tout d'abord de deux candidats, puis d'un ensemble de candidats.

1.3.1 – Deux candidats [sur 11 points]

Un croisement considère la représentation mémoire de deux candidats et la croise (= échange une sous-partie), modifiant ainsi les deux candidats :



En supposant qu'il existe une fonction

`size_t random_size_t(size_t max);`

retournant un `size_t` au hasard entre `0` inclus et `max` *exclu*, définissez ici une fonction

`void crossover_candidates(Candidate* first, Candidate* second)`

qui prend deux candidats les modifie comme suit (voir illustration ci-dessus) :

1. tirer au hasard une position `pos` entre 0 (inclus) et le nombre d'octets d'un `Candidate` (exclu) ;

2. tirer au hasard un nombre `len` d'octets entre 1 (inclus) et le nombre maximal d'octets après `pos` ; par exemple si un `Candidate` est sur 8 octets, et que `pos` vaut `3`, il faudra tirer `len` au hasard entre 1 (inclus) et 6 (exclu) ; à noter cependant le cas particulier où `pos` vaut `0` : comme on ne souhaite pas échanger complètement les deux candidats, si dans ce cas `len` vaut le nombre d'octets d'un `Candidate`, il faudra simplement réduire `len` de 1 ;
3. échanger les `len` *octets* des deux candidats à partir de l'index `pos` (inclus).

```
void swap(char* a, char* b)
{
    char c = *a;
    *a = *b;
    *b = c;
}

void crossover_implementation(Candidate* a, Candidate* b,
                              size_t pos, size_t len)
{
    // view each Candidate as an array of char
    char* const first  = (char*) a;
    char* const second = (char*) b;

    // swap middle content
    len += pos;
    for (size_t i = pos; i < len; ++i) {
        swap(&(first[i]), &(second[i]));
    }
}

void crossover_candidates(Candidate* a, Candidate* b)
{
    const size_t pos =     random_size_t(sizeof(Candidate)      );
    size_t len       = 1 + random_size_t(sizeof(Candidate) - pos);
    if ((pos == 0) && (len == sizeof(Candidate))) --len;
    crossover_implementation(a, b, pos, len);
}
```

1.3.2 – Deux ensembles de candidats [sur 4 points]

Définissez une fonction `crossover_samples()` qui prend en argument deux tableaux de `Candidates` et une taille et effectue le croisement de chacun des candidats de même rang dans ces deux tableaux (que l'on supposera chacun de taille supérieure ou égale à la taille passée en argument).

```
void crossover_samples(Candidate* males, Candidate* females, size_t nb)
{
    for (size_t i = 0; i < nb; ++i) {
        crossover_candidates(&(males[i]), &(females[i]));
    }
}
```

1.4 – Copie d'un ensemble de candidats [sur 8 points]

Dans la suite, nous aurons besoin de recopier des (sous-)ensembles de candidats vers d'autres.

Définissez ici une fonction `copy_sample()` qui ne retourne rien, mais prend en paramètres :

- l'ensemble d'origine d'où copier ;
- l'index `start` de début de copie (origine) ;
- le nombre total `nb` de candidats à copier ;
- l'ensemble d'arrivée (déjà existant dont le contenu est modifié) ;
- l'index `where`, dans l'ensemble d'arrivée, où placer le premier candidat copié.

Si `start` est plus grand que la taille de l'ensemble de départ, on ne fait rien.

Si la taille à copier (`nb`) est trop grande par rapport à `start` et à la taille de l'ensemble de départ, on la réduit à la taille maximale possible. Par exemple, si l'on demande de copier `nb=8` candidats, à partir de l'index `start=6` [2] d'un ensemble de départ de `10` candidats, il faudra réduire `nb` à `4`.

Si, depuis `where`, il n'y a pas assez de place dans l'ensemble d'arrivée pour copier tous les candidats finalement demandés, alors on ne fait rien.

```
void copy_sample(const Sample* from, size_t start, size_t nb,
                 Sample* to, size_t where)
{
    if (start > from->size) return;
    size_t end = start + nb;
    if (end > from->size) {
        end = from->size;
        nb = end - start;
    }

    if (where + nb > to->size) return; // not enough room in target

    for (size_t i = 0; i < nb; ++i) {
        to->content[where + i] = from->content[start + i];
    }
}
```

1.5 – Recherche itérative [sur 25 points]

Nous nous intéressons ici à l'étape, plus conséquente, qui consiste à gérer l'évolution pas à pas d'un ensemble de candidats en vue de maximiser la fonction `selection()` (revoir l'introduction si nécessaire).

Cette partie pourra nécessiter l'écriture de fonctions auxilaires.
En plus des fonctions précédents que vous avez définie, on supposera fournie [3] la fonction :
`void mutate_sample(Sample* s, size_t from, size_t to);`
qui remplace tous les candidats de `s` entre l'index `from` (inclus) et `to` (exclu) par leur mutation `mutate_candidate()`.

Définissez ci-contre une fonction

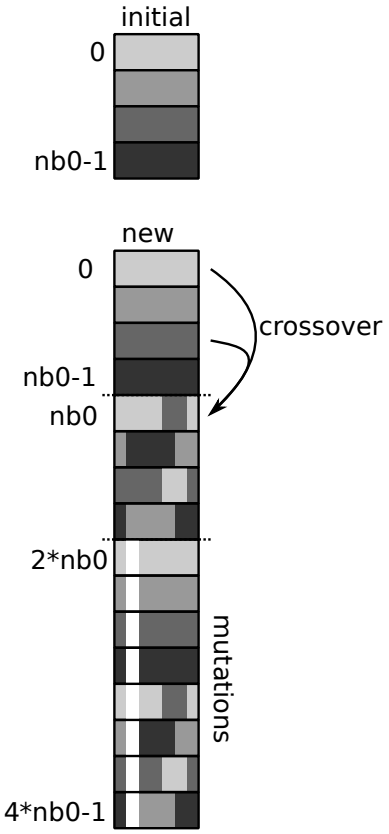
`void simulation(Sample* initial, unsigned int nb_iterations)`

qui, partant d'un ensemble initial de candidats va le modifier en un ensemble final après `nb_iterations` cycles.

2. Les index commencent bien sûr à `0`.
3. **NE PAS** l'écrire.

L'algorithme est le suivant (voir illustration ci-contre) :

- créer un ensemble de candidats, `new`, initialisé à zéro, quatre fois plus grand que l'ensemble initial (appellons `nb0` la taille de l'ensemble initial) ;
- copier l'ensemble inital au début de `new`
  (à ce stade il est donc au trois-quarts encore à zéro) ;
- pendant `nb_iterations` itérations :
  - met à partir de la position `nb0` de `new`, le croisement des `nb0` premiers candidats entre eux :
    aux positions `nb0` et `nb0 + nb0 / 2` le croisement des deux candidats qui sont aux positions `0` et `nb0 / 2` ;
    aux positions `nb0+1` et `nb0 + nb0 / 2 + 1` le croisement des deux candidats qui sont aux positions `1` et `1 + nb0 / 2` ; etc.
    il y a donc à ce stade `2 * nb0` candidats mis à jour dans `new` ;
  - ajoute à la fin de `new` la mutation de tous les candidats ci-dessus (de `0` à `2 * nb0 - 1`) ;
    autrement dit : ajoute dans `new`, à partir de `2 * nb0`, la mutation des `2 * nb0` premiers candidats de `new` ;
    il y a donc bien à ce stade dans `new`, `4 * nb0` candidats mis à jour en tout (d'où la taille de `new`) ;
  - trie le tableau `new` suivant `selection()` : de sorte à ce que les premiers candidats soient ceux qui sont les meilleurs pour `selection()` : on peut ainsi recommencer le cycle pour une nouvelle itération ;
- au final, recopie les `nb0` premiers candidats de `new` dans `initial`.




```
int compare(void const* a, void const* b)
{
    const Candidate* const p1 = a;
    const Candidate* const p2 = b;
    return selection(p1) <= selection(p2);
}
```

```
void simulation(Sample* initial, unsigned int nb_iterations)
{
    Sample new = zero_sample(4 * initial->size);

    // copy the old generation
    copy_sample(initial, 0, initial->size, &new, 0);

    const size_t half_size   = initial->size / 2;
    const size_t double_size = 2 * initial->size;

    for (unsigned int t = 1; t <= nb_iterations; ++t) {

        // creates cross over
        copy_sample(&new, 0, initial->size, &new, initial->size);
        crossover_samples(&(new.content[initial->size]),
                          &(new.content[initial->size + half_size]),
                          half_size);

        // copies the whole for mutation
        copy_sample(&new, 0, double_size, &new, double_size);
        mutate_sample(&new, double_size, new.size);

        // selection
        qsort(new.content, new.size, sizeof(*(new.content)), compare);
    }

    copy_sample(&new, 0, initial->size, initial, 0);

    delete(&new);
}
```

1.6 – Exemple d'utilisation [sur 6 points]

Voici pour finir un exemple de `main()` possible :

```
int main(void)
{
    Fit possible_functions[] = { f, g, h };
    const size_t nb = 3;

    selection = ask(possible_functions, nb);

    Sample list = random_sample(28);
    simulation(&list, 5);
    print_elements(&list, 4);

    return 0;
}
```

[3 points] Sachant que la fonction `ask()` permet de choisir un élément dans un tableau, proposez un type possible pour `Fit`. Écrivez le `typedef` correspondant :

```
typedef double (*Fit)(const Candidate*);
```

ou alors (toléré) :

```
typedef double (*Fit)(Candidate);
typedef double (*Fit)(double, double, double);
```

[3 points] Complétez si nécessaire ce `main()` (à droite avec une/des flêche(s)) ou indiquez : « rien à ajouter ».
(Toutes les fonctions utilisées ici auront été définies au préalable.)

Il manque le `delete(&list);` à la fin.