Rapport projet LO21

Léo Angonnet – Hugo Allainé

Sommaire

La compilation du programme se fait via le script de votre choix (Bash ou Batch)
Disponible dans le répertoire du projet

Description des choix de conception et d'implémentation

Pour la conception du projet, nous avons choisi d'utiliser une liste simplement chainée pour la liste de Bits composée d'un Bit et d'un pointeur vers le Bit suivant. Un pointeur vers la liste est disponible sous le terme Individu. Pour la conception de la liste d'individu, nous avons décidé au contraire d'utiliser une liste doublement chainée composée d'un individu, de sa qualité, de l'individu précédent dans la liste et du suivant. Un pointeur vers cette liste porte le nom Population. Ce choix a été fait car la fonction de tri rapide quickSort était selon nous plus simple à implémenter avec cette structure.

Dans un second temps, nous avons utilisé une version légèrement modifiée du quickSort : il faut normalement échanger de position deux éléments entre eux mais comme nous manipulons ici une liste doublement chaînée, l'implémentation aurait été compliquée en échangeant deux éléments de la liste d'individu de place car il aurait nécessité de redéfinir l'individu précédent et suivant pour chaque échange. Pour pallier ce problème, nous avons donc seulement échangé les individus et leur qualité sans toucher à leur précédent et leur suivant.

Pour créer la fonction d'initialisation de la liste de Bits constituant un individu en version récursive, nous avons trouvé une implémentation en deux fonctions : une permettant de définir le premier élément de la liste et une seconde qui définit le reste de la liste récursivement. Nous avons fait cela aussi car la condition d'arrêt de la fonction récursive se base sur un compteur qui s'incrémente à chaque appel récursif dans les paramètres de la fonction. Cela empêche l'utilisateur de pouvoir inscrire une valeur incorrecte comme compteur.

Ensuite, nous avons utilisé le module math.h pour avoir accès au fonction mathématique puissance (pow) et logarithme népérien (log), nécessaire au calcul de la qualité. Comme la qualité doit être un réel, notre choix s'est porté sur les déclinaisons réelles (float) des fonctions : powf et logf.

Dans la fonction initPopulation servant à créer une première population, nous pouvions choisir entre l'initialisation des individus de façon itérative et récursive. Nous avons donc choisi la version itérative car les fonctions itératives sont plus performantes.

De plus, nous n'avons pas utilisé de population P2 pour le croisement de la population mais nous avons réutilisé la variable de la population P1, réduisant les allocations mémoires et le code.

Enfin, les différentes variables constantes données dans l'énoncé ont été défini en tant que variable de préprocesseur. Nous avons fait ce choix pour centraliser les données et qu'elle soit facile à retrouver (disponible dans individu.h). Avec ce procédé, il est possible de changer la fonction pour calculer la qualité en mettant en commentaire la ligne 17 du fichier individu.h : en commentaire, la qualité sera calculée avec la fonction f1 de l'énoncé et décommentée elle sera calculée avec la fonction f2.

Algorithmes des sous-programmes

Sous-programme « individu.c »

Initialiser aléatoirement une liste de bits itérativement

Lexique :
individu : élément de type Individu
temp : pointeur pointant sur individu
nouvelIndividu : élément de type Individu
Résultat :
Individu

Algorithme :
```
Fonction initRandomIter() :
Début
        Allouer(individu)
        valeur(individu) ← Nombre aléatoire entre 0 et 1
        temp ← individu
        Pour i allant de 1 à longIndiv faire
        Début
                nouvelIndividu ← Individu
                valeur(nouvelIndividu) ← Nombre aléatoire entre 0 et 1
                next(nouvelIndividu) ← UNDEFINED
                Fixe le prochain élément à UNDEFINED
                next(temp) ← nouvelIndividu
                temp ← next(temp)
        Fin pour
        initRandomIter() ← individu
Fin
```

Initialiser aléatoirement une liste de bits récursivement

Lexique :
individu : élément de type Individu
nouvelIndividu : élément de type Individu
i : entier représentant un compteur
Résultat :
Individu

Algorithme :
```
Fonction initRandomRecIn(Individu individu, int i) :
Début
        Si (i == longIndiv) Alors
                initRandomRecIn ← individu
        Sinon
                Allouer(nouvelIndividu)
                valeur(nouvelIndividu) ← Nombre aléatoire entre 0 et 1
```

```
                next(nouvelIndividu) ← UNDEFINED
                next(individu) ← nouvelIndividu
                initRandomRecIn ← initRandomRecIn(next(individu), i+1)
        Fin Si
Fin
```

Initialiser aléatoirement une liste de bits récursivement

Lexique :
individu : élément de type Individu
i : entier représentant un compteur
Résultat :
initRandomRecIn

Algorithme :

```
Fonction initRandomRec() :
Début
        Allouer(individu)
        valeur(individu) ← Nombre aléatoire entre 0 et 1
        i ← 1
        initRandomRec ← initRandomRecIn(individu, i)
Fin
```

Convertir la liste de bit en une valeur en base 10

Lexique :
individu : élément de type Individu
value : entier représentant la valeur décodée de la liste de bits
i : entier représentant un compteur
Résultat :
value
Données : individu, value, i

Algorithme :

```
Fonction convertIndivToInt(Individu individu) :
Début
        value ← 0
        i ← longIndiv
        Tant que (individu ≠ UNDEFINED) Faire
        Début
                i ← i – 1
                value ← value + valeur(individu) * (2^i)
                individu ← next(individu)
        Fin tant que
        converIndivToInt ← value
Fin
```

Croiser deux listes de bits

Lexique :
individu1 : élément de type Individu
individu2 : élément de type Individu
temp1 : pointeur sur individu1
temp2 : pointeur sur individu2
temp : entier temporaire
Résultat :
Aucun (mutation)
Données : individu1, individu2, temp1, temp2, temp, pCross

Algorithme :

```
Fonction crossTwoLists(Individu individu1, Individu individu2) :
Début
        temp1 ← individu1
        temp2 ← individu2
        Tant que (temp1 ≠ NULL et temp2 ≠ NULL) Faire
        Début
                Si (Nombre aléatoire entre 0 et 100 < pCross * 100) Alors
                        temp ← valeur(temp1)
                        valeur(temp1) ← valeur(temp2)
                        valeur(temp2) ← temp
                Fin Si
                temp1 ← next(temp1)
                temp2 ← next(temp2)
        Fin Tant que
Fin
```

Calculer la qualité d'un individu à partir de sa valeur

Lexique :
value : entier représentant la valeur de l'individu
X : réel représentant une valeur dérivée de value
quality : réel représentant la qualité de l'individu
Résultat :
quality
Données : value, X, longIndiv, A, B

Algorithme :

```
Fonction calc_quality(int value) :
Début
        X ← ((value/2^longIndiv) * (B-A)) + A
        quality ← -(X^2)
        calc_quality ← quality
Fin
```

Sous-programme « population.c »

Initialiser la population avec des individus aléatoires

Lexique :
taillePop : entier représentant la taille de la population
population : élément de type Population
temp : pointeur pointant sur population
newPopulation : élément de type Population
Résultat :
population
Données : taillePop, population, temp, newPopulation

Algorithme :

```
Fonction initPopulation(int taillePop) :
Début
        Allouer(population)
        individu(population) ← initRandomIter()
        quality(population) ← calc_quality(convertIndivToInt(individu(population)))
        prev(population) ← UNDEFINED
        temp ← population
        Pour i allant de 1 à taillePop faire
        Début
                Allouer(newPopulation)
                individu(newPopulation) ← initRandomIter()
                quality(newPopulation) ← calc_quality(convertIndivToInt(individu(newPopulation)))
                next(newPopulation) ← UNDEFINED
                prev(newPopulation) ← temp
                next(temp) ← newPopulation
                temp ← next(temp)
        Fin pour
        initPopulation ← population
Fin
```

Echanger deux éléments de la population

Lexique :
a : élément de type Population
b : élément de type Population
tmp : élément de type Individu
tmp2 : réel
Résultat :
Rien (mutation)
Données : a, b, tmp, tmp2

Algorithme :

```
Fonction swap(Population a, Population b) :
Début
```

```
        tmp ← individu(a)
        individu(a) ← individu(b)
        individu(b) ← tmp
        tmp2 ← quality(a)
        quality(a) ← quality(b)
        quality(b) ← tmp2
Fin
```

Algorithme de partitionnement de liste chaînée

Lexique :

l : élément de type Population, représentant le début de la liste
h : élément de type Population, représentant la fin de la liste
i : élément de type Population, représentant un pointeur
j : élément de type Population, représentant un pointeur
x : réel représentant la valeur pivot

Résultat :

i

Données : l, h, x, i, j

Algorithme :

```
Fonction partition(Population l, Population h) :
Début
        x ← quality(h)
        i ← prev(l)
        Pour j allant de l à h faire
                Si qualité(j) ≥ x
                        i ← (i = NULL) ? l : next(i)
                        swap(i, j)
                Fin Si
        Fin pour
        i ← (i = NULL) ? l : next(i)
        swap(i, h)
        partition ← i
Fin
```

Trier une liste par ordre décroissant (version récursive)

Lexique :

l : élément de type Population représentant le premier élément de la liste
h : élément de type Population représentant le dernier élément de la liste
p : élément de type Population représentant le pivot

Résultat :

Aucun (trie la liste en paramètre)

Données : l, h, p

Algorithme :

```
Fonction quickSort(Population l, Population h) :
Début
        Si (h ≠ NULL et l ≠ h et l ≠ next(h))
```

```
                p ← partition(l, h)
                quickSort(l, prev(p))
                quickSort(next(p), h)
        Fin Si
Fin
```

Trier une liste par qualité décroissante des Individus avec la fonction quickSort

Lexique :
population : élément de type Population
start : pointeur pointant sur population
end : pointeur pointant sur l'élément de fin de population
Résultat :
population
Données : population, start, end

Algorithme :

```
Fonction sortPopByQuality(Population population) :
Début
        Si (population = NULL ou population->suivant = NULL)
                sortPopByQuality ← population
        Fin Si
        start ← population
        end ← population
        Tant que (next(end) ≠ NULL)
                end ← next(end)
        Fin Tant que
        quickSort(start, end)
        sortPopByQuality ← population
Fin
```

Sélectionner le meilleur individu de la population en tronquant la liste et en la complétant en copiant les tSelect premiers éléments.

Lexique :
population : élément de type population
tselect : entier représentant un nombre d'éléments à sélectionner
temp1 : pointeur pointant sur population
temp2 : pointeur pointant sur population
Résultat :
population
Données : population, tselect, temp1, temp2

Algorithme :

```
Fonction selectBest(Population population, int tselect) :
Début
        temp1 ← population
        temp2 ← population
        Pour i allant de 0 à tselect faire
                Si (next(temp2) = UNDEFINED) Alors
```

```
                    selectBest ← population
                Fin Si
                temp2 ← next(temp2)
        Fin Pour
        Tant que (temp2 ≠ NULL)
                individu(temp2) ← individu(temp1)
                quality(temp2) ← quality(temp1)
                temp1 ← next(temp1)
                temp2 ← next(temp2)
        Fin Tant que
        selectBest ← population
Fin
```

Créer une nouvelle population en croisant aléatoirement des individus de la population P1.

Lexique :

P1 : élément de type population
taillePop : entier représentant la taille de la population
temp1 : pointeur pointant sur P1
temp2 : pointeur pointant sur P1
rand1 : entier représentant un nombre aléatoire
rand2 : entier représentant un nombre aléatoire

Résultat :

Aucun (mutation)

Données : taillePop, temp1, temp2, rand1, rand2

Algorithme :

```
Fonction crossPop(Population P1, int taillePop) :
Début
        temp1 ← P1
        temp2 ← P1
        Pour i allant de 0 à taillePop faire
                Faire (
                        rand1 ← nombre aléatoire entre 0 et taillePop
                        rand2 ← nombre aléatoire entre 0 et taillePop
                ) Tant que (rand1 = rand2)
                Pour j allant de 0 à rand1 faire
                        temp1 ← next(temp1)
                Fin Pour
                Pour j allant de 0 à rand2 faire
                        temp2 ← next(temp2)
                Fin Pour
                crossTwoLists(individu(temp1), individu(temp2))
                quality(temp1) ← calc_quality(convertIndivToInt(individu(temp1)))
                quality(temp2) ← calc_quality(convertIndivToInt(individu(temp2)))
                temp1 ← P1
                temp2 ← P1
        Fin pour
```

Fin

Supprimer la population

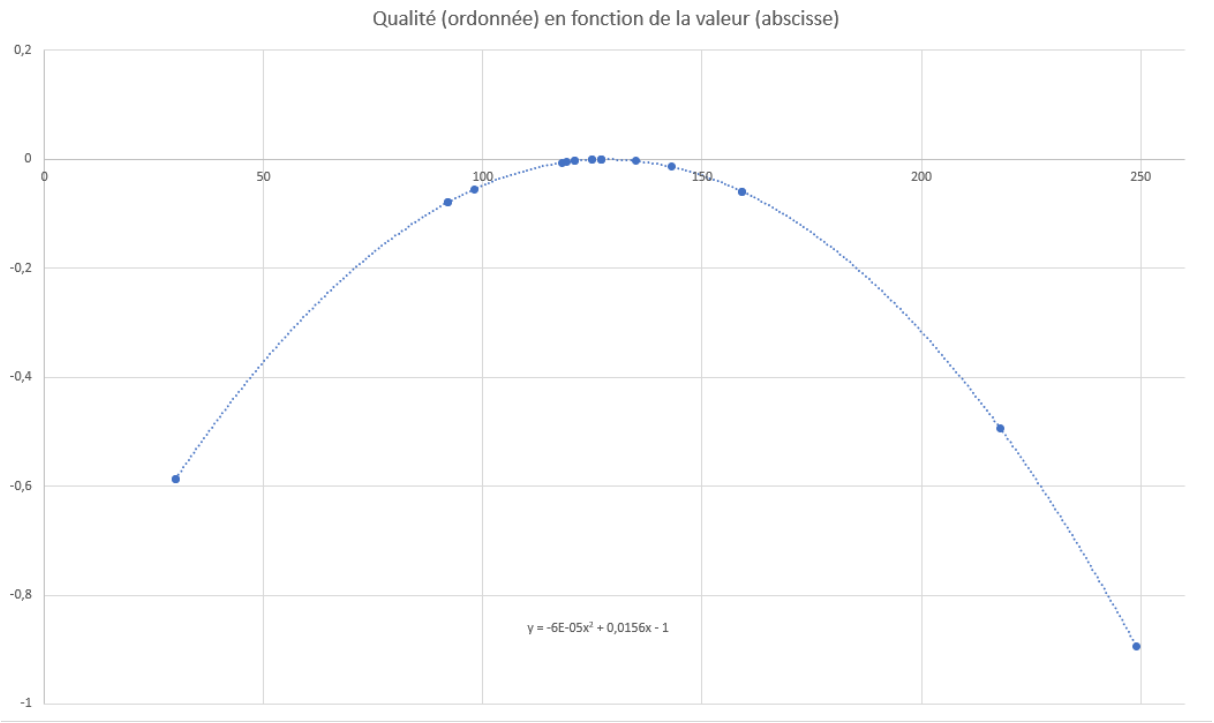
Lexique :
population : élément de type population
temp : pointeur pointant sur population
Résultat :
Aucun
Données : population, temp

Algorithme :
Fonction deletePop(Population population) :
Début
    temp ← population
    Tant que (temp ≠ NULL)
        temp ← next(temp)
        Libérer(population)
        population ← temp
    Fin Tant que
Fin

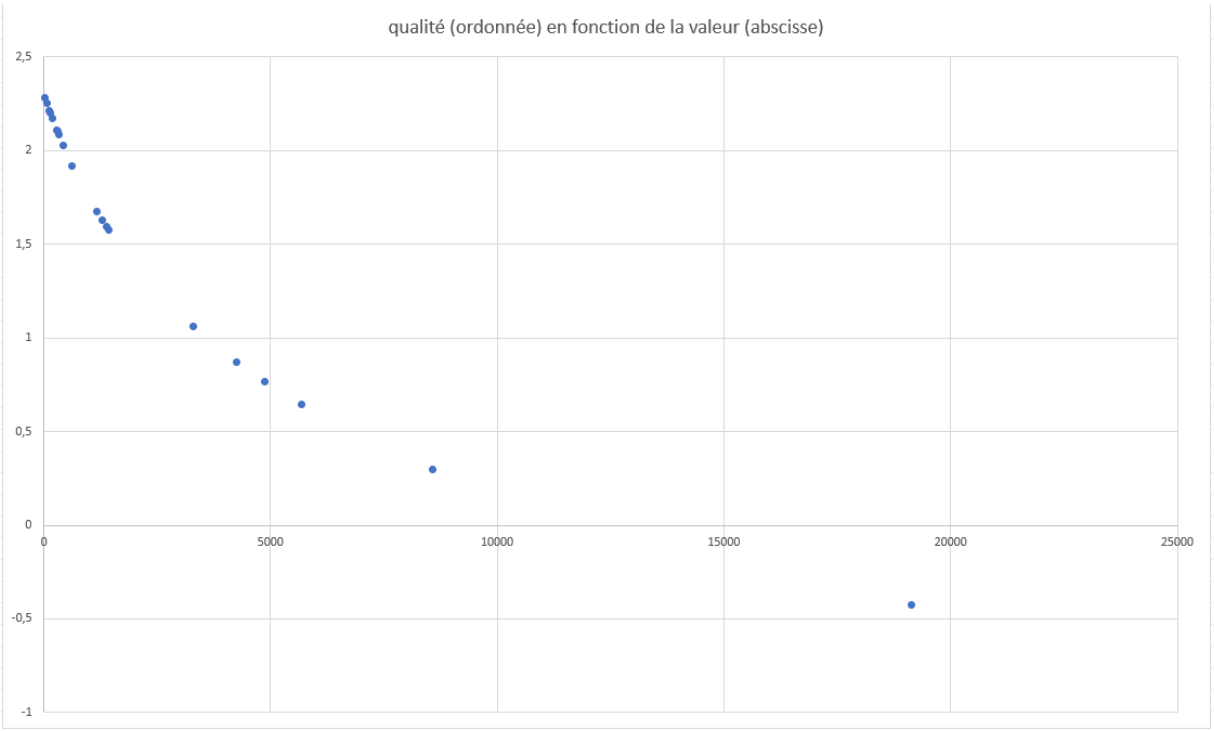
Jeux d’essais

Les jeux d’essais sont disponibles dans les fichiers textes du répertoire « Rapport »

A partir du jeu d’essais de la fonction 1, nous pouvons construire le graphique suivant :



A partir du jeu d’essais de la fonction 2, nous pouvons construire le graphique suivant :



Commentaires sur les résultats

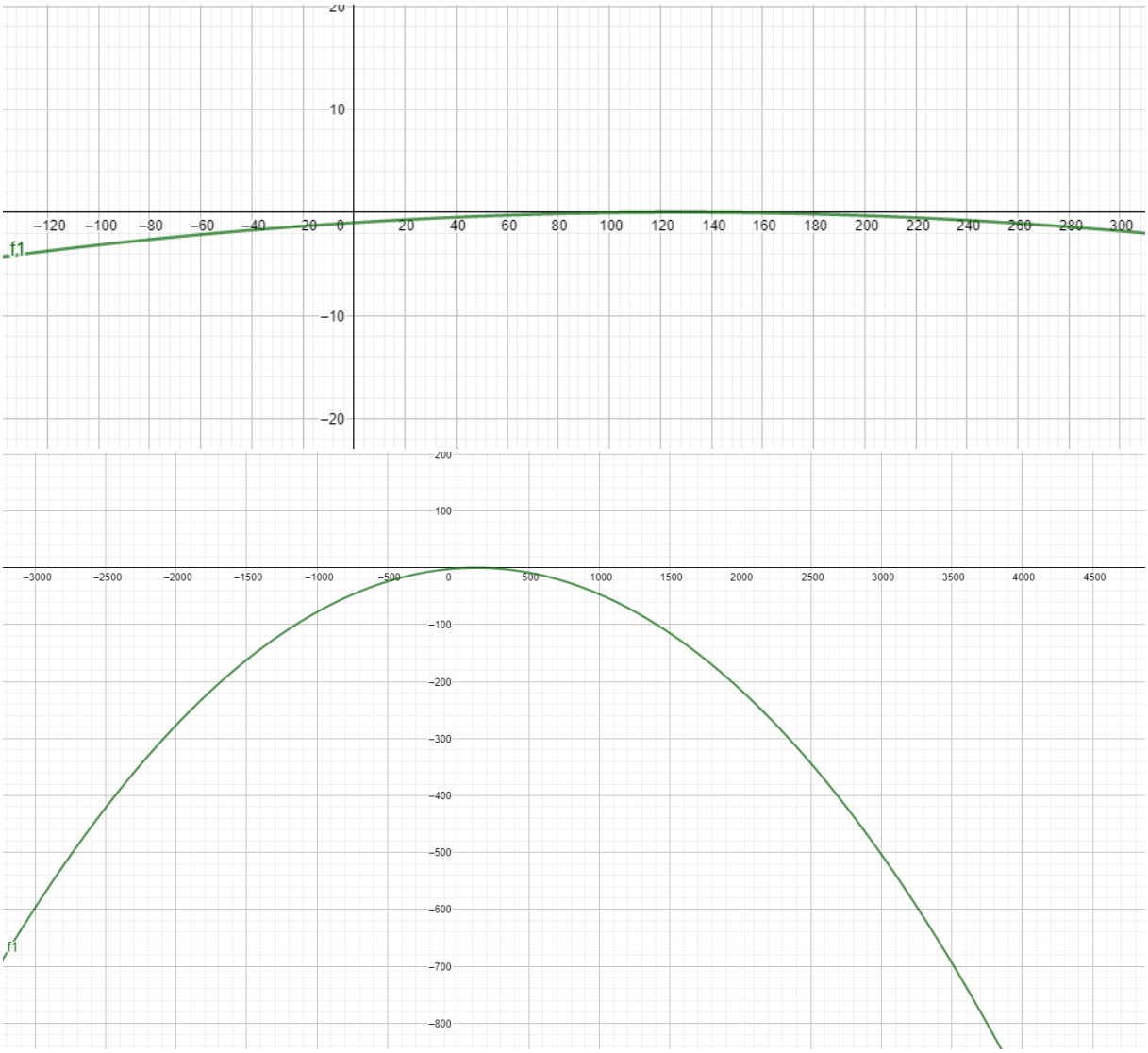
Première fonction

Pour la première fonction f1, on remarque qu'on obtient 0 en qualité pour une valeur de 128 et que plus la valeur s'éloigne de 128, positivement ou négativement, plus la qualité va baisser dans les négatifs.

On peut confirmer ces résultats théoriques avec les résultats expérimentaux (voir jeux d'essais).

```
The best individual have 242 of value and -0.793213 of quality
The best individual have 48 of value and -0.390625 of quality
The best individual have 128 of value and -0.000000 of quality
```

La fonction f1, avec la valeur en abscisse et la qualité en ordonnée.



Deuxième fonction

Pour la deuxième fonction f2, cette fois ci, on obtient environ 2,3 de qualité pour une valeur égale à 0 et la fonction est décroissante.

On peut confirmer ces résultats théoriques avec les résultats expérimentaux (voir jeux d'essais).

```
The best individual have 6352 of value and 0.553513 of quality
The best individual have 11 of value and 2.294394 of quality
```

La fonction f2, avec la valeur en abscisse et la qualité en ordonnée.