UNIVERSITÉ DE NANTES

L2 X22I020

Algorithmique et Structures de données 2

Projet

2020_2021

ATALLA Salim

L2 – Informatique_Groupe_485k

Sujet

L'objective de ce projet est de réaliser une structure de donnée *fint* pour les factorisations des entiers.

Le travail est constitué de trois branches :

1. La définition d'une SDA (Structure de Donnée Abstraite) munie des opérations.
2. La réalisation d'une SDC (Structure de Donnée Concrète) en visant les meilleures complexités possibles.
3. La programmation d'une classe *fint* en C++ en respectant les codes de la programmation orientée objets.

Les opérations de cette classe sont :

Etant donnés deux *fint* a et b et un entier strictement positif n :

- Création d'un *fint* à partir de n
- Tester si a divise b
- Calculer a × b
- Calculer a^n
- Calculer a div b
- Calculer a mod b
- Calculer a div b
- Calculer le plus grand diviseur commun : gcd (a, b)
- Calculer le plus petit multiple commun : lcm (a, b)

Remarque : le fichier source des opérations de la classe *fint* est nommé : `fint.cpp`

Fint est la liste des facteurs premiers d'un entier avec leur multiplicité, il est constitué d'un entier `nb_couples`, et d'un vecteur de couples `tab_couples`,

Un couple est un enregistrement constitué d'un grand entier strictement positif `int_t fact`, et d'un entier strictement positif `mult_t mult`.

Les types utilisés :

Type int_t = grand entier positif Type mult_t = entier positif	Type Couple = enregistrement int_t fact mult_t mult Fin

```
Classe fint = enregistrement

      // Attributs
      entier                nb_couples
      vecteur de Couple     tab_couples

      // Constructeurs
      fint (int_t d n)
      fint (liste de int_t d lf,  liste de mult_t d lm)

      // Destructeur
      ~fint ()

      // Méthodes
      fonction   to_int     () : int_t
      fonction   divides    (d a : fint) : booléen
      fonction   is_prime   () : booléen
      fonction   operator== (d a : fint, d b : fint) : booléen
      fonction   operator!= (d a : fint, d b : fint) : booléen
      fonction   lcm        (d a : fint, d b : fint) : fint
      fonction   gcd        (d a : fint, d b : fint) : fint
      fonction   operator*  (d a : fint, d b : fint) : fint
      fonction   operator/  (d a : fint, d b : fint) : fint
      fonction   pow        (d a : fint, d n : entier positif) : fint
      fonction   operator<< (os : ostream, d a : fint) : ostream

Fin
```

Variables globales :

```
MAX_INT_T : int_t // la limite autorisée pour l'objet fint
```

Remarque : au-dessous, et pour calculer la complexité temporelle, on suppose que n est la taille de la liste.

// Constructeur de l'objet fint à partir d'un entier n strictement positif

SDA :

- Signature : fint (d n : entier)
- Rôle : construire un objet fint à partir d'un entier n donné.
- Précondition : n ≥ 1
- Sortie : un objet fint contient la liste des facteurs premiers de n et la taille de cette liste.

SDC :

Complexité : $\Omega(1)$, $O(2^n + n) \in O(2^n)$

```
Variables : f : vecteur de int_t, m : vecteur de mult_t,
            taille, i : entier, couple : Couple
Début
      Si (n < 1 ou n > MAX_INT_T) alors
            ERREUR
      Sinon
            decomposition (n, f, m) // Θ(2^n)
            taille ← f.longueur
            nb_couples ← taille
            Pour i de 0 à taille-1 faire
                  couple.fact = f[i]
                  couple.mult = m[i]
                  tab_couples.push_back(couple)
            finPour
      finSi
Fin
```

// Destructeur

SDA :

- Signature : ~fint ()
- Rôle : détruire l'objet fint
- Précondition : aucune
- Sortie : aucune

SDC :

Complexité : $\Omega(1)$, $O(n)$

```
Début
      tab_couples.supprimer()
Fin
```

// Constructeur de l'objet font à partir de deux listes prés initialiser selon la
// condition de l'ordonnancement croissant de la liste des facteurs

SDA :

- Signature : fint (d lf, lm : liste de int_t)
- Rôle : construire un objet fint à partir d'une liste de facteurs et une liste des multiplicités.
- Précondition : les listes sont ordonnées selon la liste des facteurs par un ordre croissant.
- Sortie : un objet fint contient le vecteur des couples et le nombre des couples.

SDC :

```
Complexité : Ω(1), O(2n) ∈ O(n)

Variables : i, taille : entier, couple : Couple

Début

      i ← 0

      Tantque (i < lf.longueur

            et lf(i) < lf(i+1)) faire

            i ← i + 1

      finTanque

      Si (i != lf.longueur-1) alors

            ERREUR

      finSi

      taille ← lf.longueur

      nb_couples ← taille

      Pour i de 0 à taille-1 faire

            couple.fact ← lf(i)

            couple.mult ← lm(i)

            tab_couples.push_back(couple)

      finPour

Fin
```

// Décomposer un entier à ses facteurs premiers

SDA :

- Signature : procédure decomposition (d n : int_t,
  m lf : vecteur de int_t, m lm : vecteur de mult_t)
- Rôle : Décomposer un entier à ses facteurs premiers.
- Précondition : lf et lm sont vides et n > 0
- Sortie : faire des modifications sur les vecteurs lf et lm.

SDC :

Complexité : $\Omega(1)$, $O(2^n)$

```
Variables : i, j, prec : int_t, checked : booléen
Début
      i ← 2
      prec ← 0
      j ← 0
      checked ← faux
      Tantque (n ≥ 2) faire
            Tantque (n mod i = 0) faire
                  Si (i != prec) alors
                        if.push_back(i)
                        checked ← vrai
                  finSi
                  n ← n div i
                  prec ← i
                  j ← j + 1
            finTantque
            Si (checked = vrai) alors
                  lm.push_back(j)
                  checked ← faux
            finSi
            j ← 0
            i ← i + 1
      finTanque
Fin
```

// Transformation en décimale

SDA :

- Signature : fonction to_int () : int_t
- Rôle : transformer l'objet fint en décimale.
- Précondition : le nombre fint ne doit pas dépasser la limite.
- Sortie : un entier du type int_t représente le nombre fint en décimale.

SDC :

```
Complexité : Ω(1), O(m) où m est la somme des multiplicités
Variables : val, fVal, mVal : int_t
            taille, i, j : entier
Début
      val ← 1
      taille ← nb_couples
      Pour i de 0 à taille-1 faire
            fVal ← tab_couples[i].fact
            mVal ← tab_couples[i].mult
            Pour j de 0 à mVal-1 faire
                  val ← val * fVal
                  Si (val > MAX_INT_T) alors
                        ERREUR
                  finSi
            finPour
      finPour
      retourner val
Fin
```

// Tester si l'objet divise a (où a est un autre objet fint)

SDA :

- Signature : fonction divides (d a : fint) : booléen
- Rôle : tester si l'objet divise un autre objet du même type.
- Précondition : aucune
- Sortie : vrai si l'objet divise a, faux sinon.

SDC :

```
Complexité : Ω(1), O(n × m) où m est le longueur de la liste de a
Variables : i, j : entier
```

```
Début
    Si (a.nb_couples ≥ nb_couples) alors
        Pour i de 0 à nb_couples-1 faire
            j ← 0
            Tantque (j ≤ a.nb_couples
                et tab_couples[i].fact != a.tab_couples[j].fact) faire
                j ← j + 1
            finTantque
            Si (j ≥ a.nb_couples
                ou tab_couples[i].mult > a.tab_couples[j].mult) alors
                    retourner faux
            finSi
        finPour
    sinon
        retourner faux
    finSi
    retourner vrai
Fin
```

// Tester si l'objet est premier

SDA :

- Signature : fonction is_prime () : booléen
- Rôle : tester si l'objet est premier.
- Précondition : aucune
- Sortie : vrai si l'objet est premier, faux sinon.

SDC :

```
Complexité : Ω(1), O(1)
Variables : aucune
Début
    retourner nb_couple = 1
            et tab_couples[0].mult = 1
            et tab_couples[0].fact ≥ 2
Fin
```

// Les opérations :

// Opérateur ==

SDA :

- Signature : fonction operator==(d a : fint, d b : fint) : booléen
- Rôle : tester si a et b sont égaux.
- Précondition : aucune
- Sortie : vrai si a et b sont égaux, faux sinon.

SDC :

```
Complexité : Ω(1), O(n)
Variables : i : entier
Début
    Si (a.nb_couples = b.nb_couples) alors
        i ← a.nb_couples-1
        Tantque (a.tab_couples[i].fact = b.tab_couples[i].fact
            et a.tab_couples[i].mult = b.tab_couples[i].mult
            et i ≥ 0) faire
            i ← i - 1
        finTantque
        Si (i < 0) alors
            retourner vrai
        finSi
    finSi
    retourner faux
Fin
```

// Opérateur !=

SDA :

- Signature : fonction operator!=(d a : fint, d b : fint) : booléen
- Rôle : tester si a et b ne sont pas égaux.
- Précondition : aucune
- Sortie : vrai si a et b ne sont pas égaux, faux sinon.

SDC :

```
Complexité : Ω(1), O(n)
Variables : aucune
Début
    retourner !(a = b)
Fin
```

// Le plus petit commun multiple PPCM ou LCM

SDA :

- Signature : fonction lcm (d a : fint, d b : fint) : fint
- Rôle : calculer le plus petit commun multiple entre a et b.
- Précondition : aucune
- Sortie : retourner lcm(a, b).

SDC :

Complexité : Ω(1), O(n × m) ssi il n'y a aucun facteur commun entre a et b

```
Variables : v_A, v_B, v_c : vecteur de Couple
            c : Couple, nb, i, j : entier, tmp : fint
Début
    v_A ← a.tab_couples
    v_B ← b.tab_couples
    nb ← 0
    i ← 0
    j ← 0
    Tantque (v_A.longueur > i ou v_B.longueur > j) faire
        Si (v_A[i].fact = v_B[j].fact) alors
            c.fact ← v_A[i].fact
            Si (v_A[i].mult > v_B[j].mult) alors
                c.mult ← v_A[i].mult
            Sinon
                c.mult ← v_B[j].mult
            finSi
            i ← i + 1
            j ← j + 1
        Sinon
            Si (v_A[i].fact < v_B[j].fact) alors
                c ← v_A[i]
                i ← i + 1
            Sinon
                c ← v_B[j]
                j ← j + 1
            finSi
        finSi
        v_c.push_back(c)
        nb ← nb + 1
```

```
            finTantque
            tmp.nb_couples ← nb
            tmp.tab_couples ← v_c
            retourner tmp
      Fin
```

// Le plus grand commun diviseur PGCD ou GCD

SDA :

- Signature : fonction gcd (d a : fint, d b : fint) : fint
- Rôle : calculer le plus grand commun diviseur entre a et b.
- Précondition : aucune
- Sortie : retourner gcd(a, b).

SDC :

```
      Complexité : Ω(1), O(n × m)
      Variables : nb, i, j : entier, tmp : fint
                  v_c : vecteur de Couple, c : Couple
      Début
            tmp ← a
            Pour i de 0 à a.nb_couples-1 faire
                  j ← 0
                  Tantque (j ≤ a.nb_couples
                        et a.tab_couples[i].fact != b.tab_couples[j].fact) faire
                        j ← j + 1
                  finTantque
                  Si (j < b.nb_couples) alors
                        Si (a.tab_couples[i].mult > b.tab_couples[j].mult) alors
                              tmp.tab_couples[i].mult ← b.tab_couples[j].mult
                        finSi
                  Sinon
                        tmp.tab_couples[i].mult ← 0
                  finSi
            finPour
            // nettoyage le vecteur
            nb ← 0
            Pour i de 0 à tmp.nb_couples-1 faire
                  Si (tmp.tab_couples[i].mult != 0) alors
                        c ← tmp.tab_couples[i]
                        v_c.push_back(c)
```

```
                        nb ← nb + 1
                  finSi
            finPour
            tmp.tab_couples ← v_c
            tmp.nb_couples ← nb
            retourner tmp
      Fin
```

```
// Opérateur ×
```

SDA :

- Signature : fonction operator*(d a : fint, d b : fint) : fint
- Rôle : calculer la multiplication pour deux fint.
- Précondition : aucune
- Sortie : retourner a * b

SDC :

```
      Complexité : Ω(1), O(n × m) ssi il n'y a aucun facteur commun entre a et b
      Variables : v_A, v_B, v_c : vecteur de Couple
                  c : Couple, nb, i, j : entier, tmp : fint
      Début
            v_A ← a.tab_couples
            v_B ← b.tab_couples
            nb ← 0
            i ← 0
            j ← 0
            Tantque (v_A.longueur > i ou v_B.longueur > j) faire
                  Si (v_A[i].fact = v_B[j].fact) alors
                        c.fact ← v_A[i].fact
                        c.mult ← v_A[i].mult + v_B[j].mult
                        i ← i + 1
                        j ← j + 1
                  Sinon
                        Si (v_A[i].fact < v_B[j].fact) alors
                              c ← v_A[i]
                              i ← i + 1
                        Sinon
                              c ← v_B[j]
                              j ← j + 1
```

```
                finSi
            finSi
            v_c.push_back(c)
            nb ← nb + 1
        finTantque
        tmp.nb_couples ← nb
        tmp.tab_couples ← v_c
        retourner tmp
    Fin
```

```
// Opérateur div
```

SDA :

- Signature : fonction operator/(d a : fint, d b : fint) : fint
- Rôle : calculer la division pour deux fint.
- Précondition : a ≥ b
- Sortie : retourner a / b

SDC :

```
    Complexité : Ω(1) en cas d'erreur, O(2n + m + 2^k) ∈ O(2^k)
    Variables : i : entier, val_A, val_B, f : int_t, m : mult_t
    Début
        Si (a = b) alors // Θ(n) au pire a.longueur = b.longueur mais a ≠ b
            retourner fint (1)
        Sinon
            val_A ← a.to_int() // Θ(m) où m supérieur ou égale à a.longueur
            val_B ← 1
            Pour i de 0 à b.nb_couples-1 faire // Θ(n) où n égale à b.longueur
                val_B ← f^m
                val_A ← val_A div val_B
                Si (val_A < 1) alors
                    ERREUR
                finSi
            finPour
            retourner fint (val_A)  // Θ(2^k) où k est le nombre des facteurs de
                                    // l'objet créé
        finSi
    Fin
```

// Opérateur mod

SDA :

- Signature : fonction operator%(d a : fint, d b : fint) : fint
- Rôle : calculer le modulo pour deux fint.
- Précondition : aucune
- Sortie : retourner a % b

SDC :

Complexité : $\Omega(n \times m)$ en cas d'erreur, $O(n + m + 2^k) \in O(2^k)$

Variables : val_A, val_B : int_t

Début

```
Si (b.divides(a)) alors  // Θ(n × m)
    ERREUR
finSi
val_A ← a.to_int() // Θ(n) où n le somme des multiplicités de a
val_B ← b.to_int() // Θ(m) où m le somme des multiplicités de b
retourner fint (val_A % val_B)  // Θ(2^k) où k est le nombre des facteurs
                                // de l'objet créé
```

Fin

// Opérateur pow

SDA :

- Signature : fonction pow (d a : fint, d n : entier positif) : fint
- Rôle : calculer la puissance pour un fint.
- Précondition : aucune
- Sortie : retourner a^n

SDC :

Complexité : $\Omega(1)$, $O(n)$

Variables : i : entier, tmp : fint

Début

```
Si (n = 1) alors
    retourner a
finSi
tmp ← a
Pour i de 0 à a.nb_couples-1 faire
    tmp.tab_couples[i].mult ← tmp.tab_couples[i].mult * n
finPour
retourner tmp
```

Fin

Conclusion :

Ce projet a pour but de représenter les nombres entiers sous forme d'une liste contenant les facteurs premiers et leurs multiplicités,

Cette liste peut être représenter en différentes façons, par exemple :
par un tableau dynamique, un dictionnaire, ou encore une liste chaînée.

La différence entre les tableaux dynamiques et les listes chaînées est au niveau de la complexité temporelle. Les tableaux dynamiques sont idéals pour récupérer les éléments du tableau, mais ils sont complexes pour ajouter ou supprimer un élément.

Par contre, à l'inverse des tableaux dynamiques, les listes chaînées sont idéals pour ajouter ou supprimer les éléments d'une liste mais elles sont complexes pour récupérer un élément.

En regardant le sujet donné, on peut remarquer que les tableaux dynamiques sont plus compatibles avec le sujet que les listes chaînées, parce que, pour faire les opérations sur l'objet fint, on aura souvent besoin de récupérer des éléments de la liste, et donc en conséquence, on a utilisé l'objet « vector » qui représente un tableau dynamique.