Exercice 1 : Tranche maximale (structures, tableaux, pointeurs, algorithmique, niveau 2)

Positionnement du problème

On s'intéresse ici au problème suivant :
étant donnée une liste (finie) de nombres entiers, positifs et négatifs, trouver la sous-liste (contiguë) de somme maximale.

Par exemple si la liste est :

```
11, 13, -4, 3, -26, 7, -13, 25, -2, 17, 5, -8, 1
```

la sous-liste `3, -26, 7, -13, 25` a pour somme `-4`,
et la sous-liste de somme maximale est

```
    25, -2, 17, 5
```

(de somme `45`).

Un des objectifs de cet exercice est de travailler les pointeurs (qui ne sont pas strictement nécessaire pour résoudre ce problème).

Un autre objectif est de revenir sur la complexité des algorithmes et comparer différentes versions de solutions.

Structures de données

Pour résoudre informatiquement ce problème définissez les types suivants :

- `Valeur`, pour représenter les valeurs de la liste (`int` ici);
- `Liste`, pour représenter une liste de valeurs;
- `Somme_sous_liste` comme une structure comprenant :
  - un pointeur sur le premier élément d'une sous-liste (à vous de voir plus précisément);
  - un pointeur sur le dernier élément d'une sous-liste
  - la somme des valeurs correspondantes.

Pour tester, commencer de suite à déclarer dans le `main()` un tableau de listes, p.ex. :

```
vector seq({
    { 11, 13, -4, 3, -26, 7, -13, 25, -2, 17, 5, -8, 1 },
    {},
    { -3, -4, -1, -2, -3 },
    { -1, -4, -3, -2, -3 },
    { 3, 4, 1, 2, -3 },
    { 3, 4, 1, 2, 3 },
    { 3, -1, -1, -1, 5 },
    { -5, -4, 1, 1, 1 }
});
```

Algorithmes (et complexité)

Algorithme naïf

L'algorithme le plus simple qui vient à l'esprit est de chercher parmi toutes les sous-listes, celle de somme maximale. « *Chercher parmi toutes les sous-listes* » signifie, pour tous les débuts possibles, et pour toutes les fins possible pour un tel début.

D'où l'algorithme suivant :

```
Entree : liste de N nombres
Sortie : la sous-liste (debut, fin, somme) de somme maximale

sousliste = (0, 0, -infini)
Pour tout debut de 1 à N
    Pour tout fin de debut à N
        somme = 0
        Pour tout position de debut à fin
            somme = somme + liste[position]
```

```
            Si somme > sousliste.somme
                sousliste = (debut, fin, somme)
Sortir : sousliste
```

QUESTION : quelle est la complexité de cet algorithme ?

Implémentez cet algorithme (en C++) dans une fonction `tranche_max_1` (de votre programme `tranches.cc`).

Complétez la fonction `main()` pour rechercher la sous-liste (ou « tranche ») de somme maximale dans vos listes de test.

Algorithme un peu moins naïf

La complexité de l'algorithme précédent vient de ses boucles imbriquées.
Serait-il possible d'en enlever une ?

En regardant de plus prêt, on s'appercoit vite que l'on refait beaucoup de calculs plusieurs fois :
le calcul de la somme d'une sous-liste utilise souvent les calculs d'une sous-liste précédente.

En d'autres termes, la boucle la plus intérieure est inutile car pour calculer la somme à « `position+1` », il suffit d'ajouter `liste[position+1]` à l'ancienne somme.

Voici alors un nouvel algorithme :

```
Entree : liste de N nombres
Sortie : la sous-liste (debut, fin, somme) de somme maximale

sousliste = (0, 0, -infini)
Pour tout debut de 1 à N
    somme = 0
    Pour tout fin de debut à N
        somme = somme + liste[fin]
        Si somme > sousliste.somme
            sousliste = (debut, fin, somme)
Sortir : sousliste
```

QUESTION : quelle est la complexité de cet algorithme ?

Implémentez cet algorithme dans une fonction `tranche_max_2` et vérifiez qu'il fournit les bons résultats.

Algorithme linéaire

Peut-on aller plus loin ? Peut-on encore enlever une boucle ?

La réponse est **oui**, mais la solution est peut-être moins triviale.

L'idée reste cependant la même : supprimer des calculs inutiles. Mais elle utilise ici le fait que l'on cherche un maximum.

Si donc on trouve une sous-liste initiale de somme inférieure (ou égale) à 0, on peut supprimer cette sous-liste initiale car elle apporte moins à la somme totale que de commencer juste après elle.

Par exemple dans la liste `4,-5,3,...` il vaut mieux commencer au 3 (avec une somme initiale qui vaut 0) que commencer au 4 (qui nous donne une somme de -1 arrivé au 3).

D'où l'idée de l'algorithme suivant :

```
Entree : liste de N nombres
Sortie : la sous-liste (debut, fin, somme) de somme maximale

sousliste = (0, 0, -infini)
debut = 1
somme = 0
Pour tout fin de 1 à N
    somme = somme + liste[fin]
    Si somme > sousliste.somme
        sousliste = (debut, fin, somme)
    Si somme <= 0
        debut = fin + 1
        somme = 0
Sortir : sousliste
```

QUESTION : quelle est la complexité de cet algorithme ?

Implémentez cet algorithme dans une fonction `tranche_max_3` et vérifiez qu'il fournit les bons résultats.

Améliorations (FACULTATIF)

Vous pouvez aussi :

1. vous aider d'une fonction `affiche()` pour afficher les résultats ;
2. (semaine prochaine) lire des listes depuis un fichier.

Exercice 2 : dérivation formelle (pointeurs, niveau 3)

Le but est ici de faire un programme capable de dériver formellement des expressions (arithmétiques).

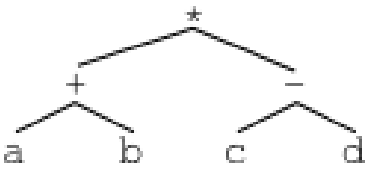
On utilisera pour cela une structure d' **arbre binaire**. Un arbre binaire est une structure de donnée définie de façon récursive par :
un arbre est

- soit vide
- soit une valeur et deux sous-arbres (binaires).

Par exemple, si l'on note un arbre entre parenthèses, en premier son sous-arbre droit, puis sa valeur et enfin son sous-arbre gauche, alors

```
(((a) + (b)) * ((c) - (d)))
```

est un arbre binaire : la valeur est `*` et les deux sous-arbres `((a) + (b))` et `((c) - (d))` :



De même `(a)` est un arbre réduit à une valeur (c.-à-d. avec 2 sous-arbres vides).

1. Dans un fichier `derivation.cc`, définissez, en vous inspirant de **l'exercice des listes chaînées** une structure de données pour représenter les arbres binaires.

   Dans cette structure de données, la valeur sera représentée par un `char`.

   Assurez-vous que l'arbre puisse partager un ou plusieurs sous-arbres. Par exemple, l'arbre `((x) + (x))` peut être construit à partir d'un seul arbre `(x)`. Pour ce faire, vous utiliserez un compteur de références c.-à-d. un nombre qui compte combiens d'arbres utilisent un sous-arbre donné (les plus avancés d'entre vous pourront utiliser ici un `shared_ptr`, hors du programme du cours).

2. Écrire la fonction permettant de créer un arbre binaire à partir d'une valeur et de deux (pointeur sur des) sous-arbres. Vous pourrez également créer une fonction qui créée ce que l'on appelle «une feuille» c'est-à-dire un arbre réduit à sa seule valeur, dont les 2 sous-arbres sont vides.
3. Écrire une fonction permettant d'afficher un arbre binaire en format parenthésé comme illustré plus haut.

   On affichera récursivement le sous-arbre de gauche, puis la valeur puis (récursivement) le sous arbre de droite.

4. Tester que son code fonctionne correctement.
5. Passons maintenant à la représentation des expressions arithmétiques.

   On utilisera pour cela les arbres binaires (nous ne considérerons ici que les opérateurs binaires `+ - / *` et ^ (puissance)) :
   par exemple `a + b` sera représenté par l'arbre
   `((a) + (b))`
   où '+' est la valeur du premier arbre et " `(a)` " et " `(b)` " ses sous-arbres.
   De même, l'expression arithmétique
   `(a + b) * (c + d)`
   sera représentée par l'arbre
   `(((a) + (b)) * ((c) + (d)))`

   Construire les arbres représentant les expressions suivantes (4 expressions) :
   x + a

(x + a) * (x + b)
((x * x) * x) + (a * x)
(x ^ a) / ((x * x) + (b * x))

Indication : commencez par créer les arbres binaires représentant les expressions `a`, `b` et `x` (seuls), puis ceux représentant les expressions `x+a`, `x+b` et `x*x`.

6. Ecrire une fonction `derive` qui prend un arbre binaire en argument, supposé être une représentation d'une expression arithmétique valide et un caractère représentant la variable par rapport à laquelle il faut dériver, et qui retourne l'arbre binaire correspondant à l'expression arithmétique dérivée.

   On procédera pour cela de façon récursive en se rappelant les règles usuelles de dérivation :

   ```
   da/dx = 0 où a est une constante (ici : caractère différent de x)

   dx/dx = 1

   d(f+g)/dx = df/dx + dg/dx

   d(f-g)/dx = df/dx - dg/dx

   d(f*g)/dx = (df/dx * g) + (f * dg/dx)

   d(f/g)/dx = ((df/dx * g) - (f * dg/dx) ) / (g * g)

   d(f^a)/dx = a * df/dx * (f ^ (a - 1))

   (où a ne dépend pas de x, ce que l'on supposera ici)
   ```

   Note importante : on ne veut pas l'arbre minimal de dérivation ! En clair, on ne cherchera pas à simplifier les expressions obtenues.

7. Testez votre fonction de dérivation sur les 4 expressions précédentes.
