MOOC Init. Prog. C++
Exercices semaine 8

Exercice 23 : Tranche maximale (structures, tableaux, niveau 1)

Positionnement du problème

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

Par exemple si la séquence est :

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

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

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

(de somme `45`).

Structures de données

Commençons par nous préoccuper des structures de données nécessaires pour résoudre informatiquement ce problème.

Il nous faut définir :

- comment représenter la séquence

  Un tableau (dynamique pour rester général) semble bien approprié

- ce qu'est une sous-séquence

  De plus il faudra connaitre la somme correspondante pour décider si elle est maximale ou non.

  Pour cela je vous propose de coder une sous-séquence et sa somme dans une structure comprenant :

  - La position de début de la sous-séquence
  - La position de fin
  - La somme correspondante

Cette première réflexion étant faite, on peut commencer.

Dans la version simple présentée ici, on définira la séquence « en dur » dans le `main()` ; mais vous êtes libres de choisir une autre solution plus avancée si vous le souhaitez.

À faire :

Dans un fichier `tranche.cc`

1. Définissez le type `Position` comme un `size_t`.
2. Définissez le type `Sequence` comme un tableau d'entiers.
3. Définisez la structure `SousSequence` contenant 2 champs, `debut` et `fin`, de type `Position`, et un champs `somme` de type entier.
4. Dans le `main()`, déclarez la variable `seq` de type `Sequence` et initialisez la une valeur choisie, par exemple :

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

Algorithme naïf

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

D'où l'algorithme de base :

```
Entrée : séquence de N nombres
Sortie : la sous-séquence (début, fin, somme) de somme maximale

soussequence = (1, 1, sequence[1])
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 + sequence[position]
    Si somme > soussequence.somme
       soussequence = (debut, fin, somme)
```

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

Complétez la fonction `main()` pour rechercher la sous-séquence (ou « *tranche* ») de somme maximale dans la séquence précédemment considérée.

QUESTION : quelle est la réponse ?

Indication

- Attention, dans les algorithmes les tableaux sont indexés de 1 à N, ce qui n'est pas le cas en C++...

Algorithme un peu moins naïf

La « lenteur » (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-séquence utilise souvent les calculs d'une sous-séquence précédente.

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

Voici le nouvel algorithme :

```
Entrée : séquence de N nombres
Sortie : la sous-séquence (début, fin, somme) de somme maximale

soussequence = (1, 1, sequence[1])
Pour tout debut de 1 à N
  somme = 0
  Pour tout fin de debut à N
    somme = somme + sequence[fin]
    Si somme > soussequence.somme
      soussequence = (debut, fin, somme)
```

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

Par exemple dans la séquence `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 :

```
Entrée : séquence de N nombres
Sortie : la sous-sequence (début, fin, somme) de somme maximale

soussequence = (1, 1, sequence[1])
debut = 1
somme = 0
Pour tout fin de 1 à N
```

```
        somme = somme + sequence[fin]
        Si somme > soussequence.somme
          soussequence = (debut, fin, somme)
        Si somme <= 0
          debut = fin + 1
          somme = 0
```

Tests

Testez vos implémentations avec les séquences suivantes :

```
-3, -4, -1, -2, -3
-1, -4, -3, -2, -3
 3, -1, -1, -1,  5
 3,  4,  1,  2, -3
 3,  4,  1,  2,  3
-5, -4,  1,  1,  1
```

Améliorations (FACULTATIF)

Vous pouvez aussi :

1. vous aider d'une fonction `affiche` pour afficher les résultats ;
2. ajouter dans les sous-séquences une référence à la séquence concernée (cf exercice 26) ;
3. faire saisir la séquence à l'utilisateur ;
4. recommencer tant que l'utilisateur le souhaite.

Exercice 24 : tri de Shell (niveau 2)

Cet exercice correspond à l'exercice n°44 (pages 98 et 288)
de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

On cherche ici à implémenter une méthode de tri assez efficace en pratique, surtout pour des tableaux de taille faible à moyenne.
Il s'agit du tri dit "de Shell" du nom de son inventeur.

Le tri de Shell est un tri par insertion à incrément décroissant : au lieu d'insérer chaque élément à sa place dans la sous-liste triée de tous les éléments qui le précèdent, on l'insère dans une sous-liste d'éléments qui le précèdent mais distants d'un certain incrément `k` que l'on fait décroître au cours du temps.

L'algorithme est le suivant (indices de 1 à taille) :

```
Pour k de taille/2 à 1, en le divisant par 2
    Pour i de k+1 à taille
        j <- i-k
        Tant que j > 0
            Si t[j] > t[j+k]
                échanger t[j] et t[j+k]
                j <- j-k
            Sinon
                j <- 0
```

Comme pour le tri bulle (exercice 1), implémentez cet algorithme en C++ (attention aux indices) et testez le sur diverses données.

Attention ! `j` peut être négatif (par `j <- j-k`)...

Exemple d'exécution

```
A trier  : 3 5 12 -1 215 -2 17 8 3 5 13 18 23 5 4 3 2 1
Résultat : -2 -1 1 2 3 3 3 4 5 5 5 8 12 13 17 18 23 215
```

Exercice 25 : culture de masse (niveau 2)

Cet exercice correspond à l'exercice n°43 (pages 97 et 282)
de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Le but est d'écrire un programme permettant d'effectuer le classement du groupe B du premier tour de l'*EuroFoot 2004* (Angleterre, Croatie, France, Suisse) après le déroulement de tous les matchs.

La liste des matchs est la suivante :

```
Suisse - Croatie
France - Angleterre
Suisse - Angleterre
France - Croatie
Angleterre - Croatie
Suisse - France
```

Pour le classement des équipes, les règles d'attribution des points sont les suivantes :

- pour chaque victoire, une équipe récolte 3 points ;
- pour chaque match nul, une équipe récolte 1 point ;
- pour chaque défaite, une équipe récolte 0 point.

Après la série de matchs, il convient de stocker dans un tableau, les informations suivantes pour chacune des équipes : points, buts marqués, buts encaissés.

Plusieurs approches sont possibles, mais il faut utiliser au moins un tableau.

Partie 1 : le calcul des points

Écrire un fichier `foot.cc` qui demande à l'utilisateur d'entrer les résultats des matchs, puis qui affiche les résultats de chaque équipe : nombre de points obtenus, nombres de buts marqués et de buts encaissés, ainsi que la différence de buts (c.f. ci-après pour un exemple de déroulement).

Partie 2 : le classement

On souhaite maintenant pouvoir afficher le classement des équipes.

Pour le classement, le choix se fera premièrement sur le nombre de points obtenus, puis en cas d'égalité sur la différence de buts, puis si nécessaire sur le nombre de buts marqués. Si malgré tous ces critères, deux équipes sont toujours à égalité, on est libre de choisir quelle équipe précède l'autre.

Définir une fonction pour comparer deux équipes, par exemple :

```
bool meilleure(const Equipe&, const Equipe&);
```

ou toute fonction similaire qui semblera convenir par rapport à ce qui a été fait jusqu'ici.

Pour effectuer le classement, on utilisera la fonction définie ci-dessus et la méthode suivante (utilisant des index à la C++ : entre 0 et 3) :

```
pour i allant de 0 à 2
  pour j allant de 3 à i+1 (en décroissant)
    si l'équipe j est meilleure que l'équipe (j-1)
    alors échanger ces 2 équipes
```

INDICATION

Il pourra être ici utile d'avoir une fonction qui échange ses deux arguments.

Exemple de déroulement

```
Suisse - Croatie ? 0 0
France - Angleterre ? 2 1
Suisse - Angleterre ? 0 3
France - Croatie ? 2 2
Angleterre - Croatie ? 4 2
Suisse - France ? 1 3

Résultats :
Angleterre : 6 points, 8 buts marqués, 4 buts encaissés, différence :
Croatie : 2 points, 4 buts marqués, 6 buts encaissés, différence : -2
France : 7 points, 7 buts marqués, 4 buts encaissés, différence : 3
Suisse : 1 points, 1 buts marqués, 6 buts encaissés, différence : -5

Le classement final est :
1 : France
2 : Angleterre
3 : Croatie
4 : Suisse
```

Exercice 26 : jeu du pendu (niveau 3)

Cet exercice correspond à l'exercice n°45 (pages 99 et 287)
de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Le jeu du « pendu » se joue à plusieurs personnes (au moins deux), chacun à tour de rôle proposant aux autres un mot à deviner.

Au départ, seule la taille du mot à deviner est connue.

Chaque joueur (sauf celui qui a posé le mot !) propose ensuite, à tour de rôle, une lettre. Si cette lettre est présente dans le mot, toutes ses occurrences sont indiquées (c.f. l'exemple de déroulement ci-après) ; sinon, celui qui a proposé la lettre « progresse » sur le chemin de la pendaison, c.-à-d. que le dessin de son pendu progresse d'une étape. Au bout de treize étapes, il a perdu et est éliminé du jeu (il est « pendu »). Précisons que le compteur de pendaison n'est jamais remis à zéro au cours de la partie.

La partie se termine lorsque tous les joueurs ont proposé un mot. Chaque joueur qui parvient à en pendre un autre gagne un point. Chaque joueur qui trouve un mot (i.e. propose la dernière lettre à trouver) gagne un point.

À la fin de la partie gagne(nt) celui/ceux qui est/sont encore en vie et a/ont le plus de points.

Exemple de déroulement

```
Nombre de joueurs :
3
Nom du joueur 1 : Pierre
Nom du joueur 2 : Paul
Nom du joueur 3 : Nathalie
Au joueur Pierre (1) de proposer un mot.
(Les autres joueurs ne regardent pas)
Entrez le mot propose : cadeau
<EFFACEMENT DE L'ECRAN>
MOT : ......
Joueur Paul (2) proposez une lettre : g
Pas de chance :

__________
MOT : ......
Joueur Nathalie (3) proposez une lettre : e
Bravo : ...e..
MOT : ...e..
Joueur Paul (2) proposez une lettre : x
Pas de chance :
  |
  |
  |
```

```
   |
__|_______
MOT : ......
Joueur Nathalie (3) proposez une lettre : a
Bravo : .a.ea.
MOT : .a.ea.
Joueur Paul (2) proposez une lettre : r
Pas de chance :
   |
   |
   |
   |
_/|_______
MOT : .a.ea.
Joueur Nathalie (3) proposez une lettre : u
Bravo : .a.eau
MOT : .a.eau
Joueur Paul (2) proposez une lettre : d
Bravo : .adeau
MOT : .adeau
Joueur Nathalie (3) proposez une lettre : c
Bravo : cadeau
-> GAGNE !
Au joueur Paul (2) de proposer un mot.
(Les autres joueurs ne regardent pas)
Entrez le mot propose :
... etc. ...
<A LA FIN :>
La partie est finie.
Resultats :
Joueur Pierre (1) encore en vie, et avec 0 point.
Joueur Paul (2) est PENDU...
Joueur Nathalie (3) encore en vie. et avec 2 points.
```

Au terme de la pendaison (13^{e} étape), le pendu pourrait ressembler à :

```
  ______
  |/ |
  | _O_
  |  I
  | / \
_/|\______
```

On peut choisir ici un affichage totalement différent. Le but n'est pas de perdre du temps à essayer de faire un joli dessin ; ce qui compte c'est que chaque joueur voit clairement à chaque tour de jeu à quel stade il en est (un décompte pourrait par exemple très bien faire l'affaire).

INDICATIONS

1. Le caractère `'\'` s'écrit `'\\'` en C++.
2. Pour « effacer » l'écran, afficher un nombre suffisant de lignes vides.
3. On peut créer une structure de données pour représenter un joueur (nom, nombre de points, étape dans la pendaison) et un tableau de tels joueurs.
4. Pour le jeu sur le mot, on peut créer une fonction prenant une lettre et deux chaînes de caractères en argument et recopiant dans la seconde chaîne la lettre passée en argument en toutes les positions où cette lettre apparaît dans la première chaîne. Cette fonction pourrait renvoyer un booléen indiquant si la lettre existe dans la première chaîne ou non.
