IFT339 - Exercices sur les tableaux, files et liste chaînées

Exercice 1.
Écrivez une version récursive de la fouille dichotomique, avec une fonction qui reçoit un tableau *tab* trié (avec accès à $tab.size()$), un élément x à trouver, et les indices *min* et *max* qui déterminent la région du tableau à chercher.

Votre code devrait se rappeler lui-même récursivement, et ne devrait pas avoir besoin d'une boucle. En classe, on a argumenté que le nombre de tours de boucle de la version non-récursive était $O(\log n)$. Ici, il n'y a pas de boucle, alors comment peut-on analyser le temps O de la fonction?

Solution.

```
 int fouilleDichoRec(vector<int>& tab, int x, int min, int
max){
     if (min > max)
         return -1;
     int mid = (max - min)/2 + min;
     if (x < tab[mid])
         return fouilleDichoRec(tab, x, min, mid - 1);
     else if (x > tab[mid])
         return fouilleDichoRec(tab, x, mid + 1, max);
     else
         return mid;
 }
```

On voit qu'un appel à la fonction prend un temps $O(1)$ (si on ne compte pas les appels récursifs). La complexité est donc directement proportionnelle au nombre d'appels récursifs effectués. C'est ce qu'on doit analyser au lieu du nombre de tours de boucle, mais l'argument est le même.

Au premier appel, l'espace de recherche est de taille n. Au deuxième appel, il est de $n/2$, puis $n/4$, et ainsi de suite jusqu'à ce qu'il reste 1 élément à chercher. Le nombre d'appels est donc le nombre de fois qu'il faut diviser n par 2 pour obtenir 1, qui est $O(\log n)$ tel que vu en classe.

Exercice 2.
Vous voulez trier les personne étudiantes de votre classe en ordre croissant de leur note en IFT339. Une personne est représentée par une struct

```
struct Etudiant{
    string nom;
    string prenom;
    string cip;
    uint32_t note_IFT339;
};
```

où $note\_IFT339$ est un entier entre 0 et 100. Notez que pour bien faire les choses, cette classe devrait s'appeler PersonneEtudiante, mais c'est plus long à écrire.

Écrivez une fonction qui reçoit un vecteur de Etudiant, et qui retourne un autre vecteur du même type, dans lequel les personnes sont triées en ordre non-décroissant de leur note en IFT339 (si deux personnes ont la même note, leur ordre relatif n'est pas important). Le vecteur retourné devrait contenir des copies des personnes.

Solution.

Si on utilise std::sort ou autre fonction de tri standard, on aura un temps $O(n \log n)$. Pour faire mieux, on se fait 101 vecteurs, un pour chaque note

possible, on place les personnes dans le bon vecteur, et on les défile dans l'ordre.

Pour optimiser un peu, je mets des pointeurs dans ce vecteur de vecteurs temporaire, pour éviter les copies (ce n'était pas absolument nécessaire). Les copies à faire se font dans la dernière boucle.

```
vector<Personne> sort339(vector<Etudiant>& etuds){
    vector<vector<Etudiant*>> etud_par_note(101);
    for (auto it = etuds.begin(); it != etuds.end(); ++it){
        Etudiant& et = *it;
        etud_par_note[et.notes_IFT339].push_back(&et);
    }

    vector<Personne> sortie;
    for (int i = 0; i <= 100; ++i){
        for (int j = 0; j < etud_par_note[i].size(); ++j){
            sortie.push_back(etud_par_note[i][j]);
        }
    }
    vector<Etudiant> sortie;
}
```

La première boucle se fait clairement en temps $O(n)$, et remplis l'objet etud_par_note avec un total de n éléments. La deuxième boucle ne fera en fait que repasser à travers ces n éléments pour les insérer dans la sortie, et elle prend donc un temps $O(n)$ aussi.

Exercice 3.
Donnez le code pour la fonction erase_batch de la classe vector. Cette fonction reçoit un argument vector<size_t> pos, qui est une liste **triée** de positions à effacer dans le vecteur. Visez $O(n)$, où n est la taille du vecteur courant.

Par exemple, si *tab* contient [1 5 2 9 7 2] et que *pos* = [2 4], alors après l'appel *tab* contiendra [1 5 9 2].

Solution.

L'idée est de parcourir tab de gauche à droite et écraser $tab[i]$ par la valeur qui devrait s'y trouver après les suppressions. La valeur de $tab[i]$ devrait être $tab[i+k]$, où k indique les nombre de suppressions faites jusqu'à l'indice i.

C'est un peu laborieux de retrouver ce k. Ce qu'on peut faire, c'est garder trois index: tab_curindex indique l'indice de la prochaine case qu'on doit écraser avec la valeur correcte; tab_nextindex indique l'indice contenant la prochaine valeur non-supprimée; pos_index indique où on est dans le tableau des positions à supprimer. Quand tab_nextindex rencontre un indice présent dans *pos*, on doit ignorer l'élément $tab[tab\_nextindex]$ et l'incrémenter. On incrémente aussi pos_index. Sinon, on copie $tab[tab\_nextindex]$ à la bonne place $tab[tab\_curindex]$ et on avance.

```
void erase_batch(vector<size_t>& pos){
    int tab_curindex = 0;
    int tab_nextindex = 0;
    int pos_index = 0;

    while (tab_nextindex < nbelem){

        if (pos_index < pos.size() - 1 &&
                tab_nextindex == pos[pos_index]){
            tab_nextindex++;
            pos_index++;
        }
        else{
            tab[tab_curindex] = tab[tab_nextindex];
            tab_curindex++;
            tab_nextindex++;
        }

    }

    nbelem -= pos.size();

}
```

Exercice 4.
Reconsidérez l'exercice précédent, mais dans le cas où la liste *pos* n'est **pas triée**. Pouvez-vous tout de même atteindre un temps de $O(n)$?

Solution.

Vous pourriez utiliser std::sort, mais ceci prendra un temps $O(m \log m)$, où $m = pos.size()$. Si $m < n$, ceci peut être avantageux, mais si m est proche de n, ceci sera proche de $n \log n$, alors que $O(n)$ est possible.

Sachant que les positions sont entre 0 et $n-1$, on peut les trier d'abord en indiquant quelles positions sont présentes dans un tableau de bool (ou autre), puis en reparcourant ce tableau.

Donc si dans la fonction de l'exercice précédent, *pos* n'est pas trié, on fait un pré-traitement comme suit.

```
vector<bool> pos_presente(nbelem, false);
for (int i = 0; i < pos.size(); ++i){
    //on ne gere pas les erreurs si pos[i] > nbelem
    pos_presente[ pos[i] ] = true;
}

vector<size_t> pos_triee;
for (int i = 0; i < nbelem; ++i){
    if (pos_presente[i])
        pos_triee.push_back(i);
}

//puis travailler avec pos_triee
```

Exercice 5.
Vous maintenez le compte d'une population infectée par un virus. On suppose que la population est infinie. Au début de chaque jour, s'il y a p personnes infectées, il y aura p nouvelles personnes infectées à la fin de la journée (une

façon de voir est que chaque personne infectée va en infecter une autre dans la journée). Toutefois, toute nouvelle personne infectée guérira dans k jours, et ne sera plus infectée.

Par exemple, si $k = 3$ et que Robert se fait infecter au jour 5, alors aux jours $6, 7, 8$, Robert infectera une nouvelle personne, puis ne sera plus sur la liste des infectés au jour 9.

Étant donné un nombre d'infectés p, le nombre de jours de guérison k, et un entier n, donnez un algorithm qui calcule le nombre de personnes infectée après n jours. Quelle est votre complexité?

Solution.

On peut utiliser une file de taille k, où la position i représente le nombre de personnes qui vont guérir dans k jours. On maintient aussi le compte total.

```
int total = p;
deque<int> infect;
//initialement au jour 0, il y a p personnes qui gueriront
dans k jours
for (int i = 0; i < k - 1; ++i){
    infect.push_back(0);
}
infect.push_back(p);

//on itere les journees j de 1 a n
for (int j = 1; j < n; ++j){
    infect.push_back(total);    //nouveaux infectes
    total += total;
    total -= infect.front();
    infect.pop_front(); //ont gueri
}
cout<<total;
```

Exercice 6.
Implémentez l'opérateur == d'une liste simplement chaînée (deux listes sont égales si tous leur éléments aux positions correspondantes sont égaux). Cet opérateur reçoit une autre liste chaînée.

Solution.

On suppose une classe liste avec accès à Noeud, comme vu en classe. Il faut itérer à travers les noeuds et vérifier que chacune est égale. Il faut aussi vérifier que le nombre d'éléments est le même.

```
template <typename TYPE>
class list{
private:
    struct Noeud{
        TYPE elt;
        Noeud* next;
    };
    Noeud* debut;
    Noeud* fin;
    //...
};

template <typename TYPE>
bool list<TYPE>::operator==(const list& autre) const
{
    Noeud* p1 = this->debut;
    Noeud* p2 = autre.debut;
    if (!p1 && !p2)
        return true;    //2 listes vides = ok
    while (p1 && p2)    //on avance sur les 2 listes
    {
        if (p1->val != p2->val)
            return false;
        p1 = p1->next;
        p2 = p2->next;
    }
```

```
    if (p1 && !p2 || p2 && !p1)
        return false;   //si une liste est plus courte
    return true;
}
```
