IFT339 - Exercices sur la complexité O et les listes

Exercice 1.

(a) Pour s'échauffer tranquillement, considérez le code suivant.

```
std::vector<int> vec(10);
vec[0] = 0;
int x = vec[0];
x = 10;
std::cout << vec[0] << std::endl;
```

Dites quelle sera la sortie affichée et expliquez pourquoi.

(b) Si l'intention était de manipuler x tout en modifiant le contenu du vecteur, il serait peut-être plus utile d'utiliser int& x = vec[0];. Ceci est problématique. Considérez le code suivant.

```
std::vector<int> vec(10);
vec[0] = 0;
int& x = vec[0];

for (int i = 0; i < 10000; ++i)
    vec.push_back(i);
x = 10;
//...
```

Ceci cause parfois un crash quand je l'exécute (et parfois non, en tout cas, il y a un problème!). Expliquez ce qui ne va pas avec ce code.

(c) Proposez une modification à l'implémentation du vector pour éviter ce problème.

Solution.

Pour le (a), la sortie sera "0", puisque x est une copie de vec[0] et donc x = 10 ne change rien dans le vecteur.

Pour le (b), il se peut que ça ne crash pas, ça dépend des machines et autres facteurs. Le problème est que x est une référence vers l'entrée mémoire de vec[0]. La série de push_back détruit le tableau initial du vecteur et le recréé ailleurs. Toutefois, x est encore une référence vers la vieille zone mémoire, qui a été supprimée et qui ne nous appartient plus. Si cette zone mémoire est utilisée pour autre chose, faire $x = 10$ peut avoir des conséquences assez graves (certains exploits de sécurité utilisent d'ailleurs ce style de bug).

Pour le (c), on peut redéfinir la représentation du vecteur pour que chaque élément de *tab* soit un pointeur, et donc chaque élément serait alloué dynamiquement. Le type de tab serait donc TYPE** tab, et il faudrait initialiser et détruire chaque entrée, par exemple avec

```
tab = new TYPE[cap];
for (int i = 0; i < cap; ++i)
    tab[i] = new TYPE();
```

Il ne faut pas oublier de faire les delete tab[i] dans le destructeur! L'opérateur $[\,](size_t i)$ devrait retourner *(tab[i]).

L'avantage maintenant est que le "reserve" n'a pas à recopier les valeurs, mais plutôt les pointeurs. Les zones mémoire pointées restent inchangées, ce qui fait que le x continuera de pointer au bon endroit.

Un des désavantages est que les éléments ne seront plus stockés de façon contigue — il seront répartis aléatoirement dans la mémoire.

Exercice 2.
Considérez une alternative à l'implémentation vector vue en classe. On stocke tab un tableau de tableaux, et la classe a une variable membre subsize qui définit le nombre d'éléments à stocker dans chaque sous-tableau. Donc, tab[0] stocke les subsize premiers éléments, tab[1] les subsize suivants, et ainsi de suite.

```
template <typename TYPE>
class vector{
```

```
private:
    TYPE** tab;
    size_t nbelem;
    size_t subsize; //taille des sous-tableaux
    size_t nbtabs;  //taille de tab
    //...autres?
}
```

On veut supporter les fonctions operator[ ], push_back, pop_back. Imaginez comment les implémenter, sans nécessairement le faire. Est-ce que les variables membres suggérées sont suffisantes? Justifiez votre réponse.

Donnez ensuite la complexité en temps de ces fonctions, en fonction de $n = nbelem$ et $m = subsize$. Justifiez vos réponses.

Solution.

L'idée est qu'un débordement de push_back n'aura besoin que de détruire et recréer le premier niveau de tableau *tab*, mais pas les sous-tableaux. Par ex, pour ajouter une capacité de *subsize*, on ajoute un sous-tableau via

```
    TYPE** temp = new TYPE[nbtabs + 1];
    for (int i = 0; i < nbtabs; ++i)
        temp[i] = tab[i];   //copie des pointeurs
    temp[nbtabs] = new TYPE[subsize];

    delete [] tab;
    tab = temp;
    nbtabs++;
```

Ceci prend un temps $O(subsize) = O(m)$, dans le pire cas.

Pour ajouter un élément *val*, on le met dans $tab[nbtabs - 1][0] = val$. Il manque toutefois une variable membre, pour savoir combien de cases du dernier sous-tableau sont remplies. On peut ajouter par exemple une variable *last_index*.

Le *pop_back* peut se faire en décrémentant cette variable *last_index*, en temps $O(1)$. Il faut gérer le cas spécial où on revient sur le sous-tableau $tab[nbelem - 2]$, soit en supprimant le sous-tableau maintenant inutile (en temps $O(m)$), ou bien en le laissant là et en se rappelant du nombre de sous-tableau utilisés (avec un autre variable membre).

L'opérateur [ ] est facile à implémenter en temps $O(1)$ avec la variable $last\_index$ bien gérée, il faut retourner $tab[nbtabs - 1][last\_index]$.

Exercice 3.
Donnez la notation O des fonctions suivantes.

a) $50 + 250n + 3n^3$

b) $0.00001n^2 + 500000n$

c) $2n^2 - 10000n$

d) 10^{100}

e) $2^n + n^{10}$

f) $2n \cdot \log n + 100n$

g) $2n \cdot \log n \cdot \log n + 100n^3$

h) $50n + 2m^2 + 10m$

i) $50n^2 + 10nm + 4n \log m + m^2$

Solution.

(a) $O(n^3)$
(b) $O(n^2)$
(c) $O(n^2)$ (un terme négatif se fait dominer par n'importe quoi)
(d) $O(1)$
(e) 2^n, une fonction exponentielle finira toujours par dominer une fonction polynomiale
(f) $O(n \log n)$
(g) $O(n^3)$
(h) $O(n + m^2)$
(g) $O(n^2+m^2)$, ou encore j'accepterais $O(n^2+nm+m^2)$, car ce n'est pas si évident que nm se fera dominer soit par n^2, soit par m^2.

Exercice 4.

Pour chacun des segments de code suivants, donnez leur complexité en temps avec la notation O, en fonction de n (et de m si présent).

a)

```
int* tab = new int[n];
for (int i = 0; i < n; i++)
{
    tab[i] = i;
}
for (int i = n - 2; i >= 0; --i)
{
    tab[i] += tab[i + 1];
}
delete [] tab;
```

Solution.

On a juste deux boucles en temps linéaire, c'est $O(n)$. Notez que même si on avait 1000 boucles séparées qui font un nombre linéaire de tours, ce serait toujours $O(n)$.

---

b)

```
vector< deque<int> > tab(10);
for (int i = 0; i < tab.size(); ++i)
{
    for (int j = 0; j < n; ++j){
        tab[i].push_front(i + j);
    }
}
```

Solution.

On fait une série de n push_front, dans 10 tableaux différents. Sachant que le push_front a les mêmes garanties que le push_back, le temps pour faire n push_front est d'ordre $c \cdot n$ pour une constante n. On répète ça 10 fois, pour un temps proportionnel à $10 \cdot c \cdot n$, ce qui est $O(n)$.

c) En utilisant fouilleDicho tel que vu en classe.

```
int* tab = new int[n];
for (int i = 0; i < n; i++)
{
    tab[i] = 2 * i;
}

for (int i = 0; i < n; i++)
{
    if (fouilleDicho(tab, n, i))
        cout<<"OUI"<<endl;
    else
        cout<<"NON"<<endl;
}
delete [] tab;
```

Solution.

On exécute n fois fouilleDicho, qui prend un temps $O(\log n)$. Si on inclut le temps de la première boucle, le temps est de l'ordre $n + n \log n$, donc $O(n \log n)$.

---

d)

```
void init_tab(int* t, int n)
{
    for (int i = 0; i < n; i++)
        t[i] = 0;
}

int** tabs = new int*[n];
for (int i = 0; i < n; i++)
{
    tabs[i] = new int[m];
}

for (int i = 0; i < n; i++)
```

```
{
    initTab(tabs[i], m);
}

for (int i = 0; i < n; ++i)
    delete [] tabs[i];
delete [] tabs;
```

Solution.

L'initialisation des tab[i] est de l'ordre de n. Puisque $initTab$ est en temps linéaire par rapport à son entrée, chaque appel à $initTab$ prend un temps de l'ordre de m. On répète ça n fois, et c'est donc $O(nm)$.

---

e)

```
vector< vector<double> > v(n);
for (int i = 0; i < v.size(); ++i){
    for (int j = 0; j < m; ++j){
        v[i].push_back( rand() );   //temps O(1)
    }
}
for (int i = 0; i < v.size(); ++i){
    std::sort(v[i].begin(), v[i].end());
}
```

On commence avec n répétitions de m push_back, en temps $O(nm)$. La deuxième boucle fait n tris, chacun sur m éléments. Si on regarde la doc de std::sort, on voit qu'elle prend $O(m \log m)$. Le temps de la deuxième boucle est donc de l'ordre de $n \cdot m \log m$. Ceci domine la première boucle et on a donc $O(nm \log m)$.

Exercice 5.
Écrivez un code ou pseudo-code pour faire la recherche d'un élément dans un tableau circulaire **trié** en temps $O(\log n)$ (voir le devoir 2). Vous avez accès aux pointeurs

debut_cap, fin_cap, debut_elem, fin_elem. Vous pouvez utiliser tous les concepts vus en classe.

Solution.

On peut facilement imaginer une version de
fouilleDicho(TYPE* tab, size_t n, TYPE& k)
qui reçoit un pointeur vers le début d'un tableau, la taille du tableau, et qui effectue la fouille dichotomique en temps $O(\log n)$ sur le sous-tableau spécifié. On peut utiliser fouilleDicho sur notre tableau circulaire en lui passant les sous-tableau appropriés.

```
bool fouilleCirc(TYPE* pDebutCap, TYPE* pFinCap, TYPE* pFront, TYPE*
   pBack, TYPE& k)
{
    if (pFront < pBack)
        return fouilleDicho(pFront, pBack - pFront + 1, k);
    else
        if (k <= *pFinCap) //k est dans la partie droite
            return fouilleDicho(pFront, pFinCap - pFront + 1, k);
        else    //k est dans la partie gauche
            return fouilleDicho(pDebutCap, pBack - pDebutCap + 1, k);
}
```

Puisque cet algorithme exécute un nombre constant d'instructions et fait un seul appel à fouilleDicho, la complexité sera de $O(\log n)$.
