# Exercice : Election dans une liste chaînée circulaire

1. **Proposer un type explicite pour représenter une liste simplement chaînée circulaire.**
2. **Pour l’exercice, il n’est pas nécessaire que le type que vous avez choisi puisse représenter une liste vide, pourquoi ?**

Il n’est pas nécessaire que le type choisi puisse représenter une liste vide car, s’il n’y a pas de candidats, il n’y a pas d’élection. Cette liste n’a donc pas d’intérêt sans valeurs.

1. **Implémenter l’algorithme d’élection proposé. Il doit renvoyer le candidat élu et afficher à l’écran les candidats éliminés au fur et à mesure. (L’affichage pourra être commenté lorsque vous ferez des tests sur de grandes instances.)**
2. **Tester sur l’exemple donné ci-dessus**
3. **Faire un programme qui affiche le candidat élu pour k qui vaut 2, 3, 5 puis 10 et dans les cas où 103 104, 105, puis 106 candidats se présentent.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **103** | **10⁴** | **10⁵** | **10⁶** |
| **2** | 977 | 3617 | 68929 | 951425 |
| **3** | 604 | 2692 | 92620 | 637798 |
| **5** | 763 | 646 | 40333 | 718997 |
| **10** | 63 | 9143 | 77328 | 630538 |

1. **Etudier le coût de façon exacte puis en utilisant une notation de Landau, en précisant s’il s’agit d’un pire cas, meilleur cas ou cas moyen. Expliquer le résultat.**

**Indication : pour vous aider, introduisez dans votre algorithme une variable qui compte le nombre de tours de boucle et faites des tests systématiques, par exemple pour n = 100 et k variant de 1 à 200.**

# Problème : comparaison de tris simples

**Il s’agit d’implémenter puis de tester trois algorithmes de tris (non récursifs) sur des tableaux d’entiers afin de pouvoir comparer leurs performances.**

**Ces trois algorithmes ont été largement étudiés et il en existe de multiples implémentations sur internet.**

**Vous pouvez reprendre ce que bon vous semble à condition de citer vos sources proprement ! Il vous est demandé par contre de bien avoir compris le principe de chacun des algorithmes.**

## 3.1 Tri par insertion dichotomique

**Le premier algorithme est un tri par insertion, comme présenté en cours, mais améliorée de la façon suivante : à chaque fois qu’un élément doit être inséré dans la tranche des éléments triés, la recherche de la position où insérer l’élément se fait grâce à une recherche dichotomique.**

## 3.2 Tri bulle mélangé (shaker sort)

**Le deuxième algorithme est une variante du tri bulle. Le tri bulle pourrait s’implémenter par cette fonction :**

Void triBulle(array(i n t) RW t) {

f o r( i n t i = 0; i< t.length; i++)

f o r( i n t j = t.length - 1; j > i; j--)

i f (t[j] < t[j - 1])

swap(t, j - 1, j);

}

**Le tri bulle mélange ou *shaker sort* fonctionne comme le tri bulle en alternant les passes de gauche à droite et de droite à gauche.**

## **3.3 Shell sort**

**Le troisième algorithme, le *shell sort* est une variante du tri par insertion qui permet des permutations entre des éléments éventuellement non adjacents du tableau. Voici comment fonctionne ce tri.**

**Soit *k* une constante entière positive plus petite que la taille du tableau. On considère le sous-ensemble des éléments à trier composé des éléments d’indices 0, *k*, *2k* ,3*k*… et on applique le principe du tri par insertion (sans dichotomie), en place, sur ces éléments. (Cela revient à faire + *k* pour aller à l’indice suivant, au lieu de + 1 dans le tri par insertion.)**

**On applique ensuite ce principe successivement sur les éléments d’indices 1, 1+k, 1+2k, 1+3k…, puis 2, 2+k, 2+2k, 2+3k…, etc. jusqu’à (k-1), (k-1)+k, (k-1)+2k, (k-1)+3k…**

**(Bien entendu, pour *k* valant 1, on retrouve l’algorithme de tri par insertion.)**

**L’algorithme de *shell sort* est l’application répétée de cette procédure en suivant une séquence de valeurs de *k* bien choisie.**

## **3.2 Travail à faire**

### **Question 2**

1. **Implémenter l’algorithme correspondant.**
2. **Faire des tests fonctionnels simples.**
3. **Mettre en place un moyen de compter le nombre de comparaisons et le nombre d’affectations réalisée, et un moyen de mesurer le temps d’exécution.**

### Question 3

1. **Implémenter une fonction qui remplit aléatoirement et uniformément un tableau de taille fixée de façon à se rapprocher de l’hypothèse d’équirépartition utilisée pour les coûts en moyenne.**
2. **Comparer les trois solutions sur des tailles croissantes d’instance, en termes de nombre d’affectations, nombre de comparaisons et temps d’exécution.**
3. **Pour le *shell sort*, on pourra utiliser les séquences de valeurs de *k* suivantes : 1, 4, 13, 40, 121, 364, 1093, 3280, 9841…, ou 1, 8, 23, 77, 281, 1073, 4193, 16577…, ou encore 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81, …1**
4. **Mettre les résultats sous forme graphique ! Et commenter brièvement les résultats remarquables.**

### Question 4

1. **Pour chaque tri, donner quelques arguments simples qui expliquent pourquoi l’algorithme fonctionne (en quoi est-ce que c’est trié ?).**
2. **En particulier, pour le tri bulle mélangé, commencer par étudier des invariants de boucle du tri bulle (simple), puis s’en servir pour expliquer sa variante.**
3. **En particulier, pour le *shell sort*, quelle propriété simple doit avoir la séquence des valeurs de *k* pour garantir le tri ?**

### Question 5

1. **Etudier le coût en temps de chacun des algorithmes. Bien préciser si vous parlez de pire cas, de meilleur cas ou de cas moyen. Pour un pire cas ou un meilleur cas, il faut préciser possible sur quelle instance il se manifeste.**
2. **Pour le tri par insertion avec dichotomie, identifier si la dichotomie fait gagner en performance et si oui en quoi.**
3. **Comparer les résultats théoriques obtenus avec les performances mesurées précédemment.**

### Question 6

1. **Le tri par insertion dichotomique améliore-t-il le tri par insertion ?**
2. **Le tri bulle mélangé améliore-t-il le tri bulle ?**
3. **Le *shell sort* améliore-t-il le tri par insertion ?**