COURS D'ALGORITHME AVANCE

CHAP 1. LES TABLEAUX

« *Si on ment à un compilateur, il prendra sa revanche.* » - Henry Spencer.

Bonne nouvelle ! Je vous avais annoncé qu'il y a avait en tout et pour tout quatre structures logiques dans la programmation. Eh bien, ça y est, on les a toutes passées en revue.

Mauvaise nouvelle, il vous reste tout de même quelques petites choses à apprendre…

1. Utilité des tableaux

Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions « Lire » distinctes, cela donnera obligatoirement une atrocité du genre :

```
Moy ← (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12)/12
```

Ouf ! C'est tout de même bigrement laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, alors là c'est le suicide direct.

Cerise sur le gâteau, si en plus on est dans une situation on l'on ne peut pas savoir d'avance combien il y aura de valeurs à traiter, là on est carrément cuits.

C'est pourquoi la programmation nous permet **de rassembler toutes ces variables en une seule**, au sein de laquelle chaque valeur sera désignée par un numéro. En bon français, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ». C'est largement plus pratique, vous vous en doutez.

Un ensemble de valeurs portant le même nom de variable et repérées par un nombre, s'appelle un tableau, ou encore une variable indicée.
Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle – ô surprise – l'indice.
Chaque fois que l'on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément, entre parenthèses.

2. Notation et utilisation algorithmique

Dans notre exemple, nous créerons donc un tableau appelé Note. Chaque note individuelle (chaque emplacement du tableau Note) sera donc désignée Note[0], Note[1], etc. Eh oui, attention, les indices des tableaux, par convention, commencent généralement à 0, et non à 1.

Un tableau doit être déclaré comme tel, en précisant le nombre et le type de valeurs qu'il contiendra (la déclaration des tableaux est susceptible de varier d'un langage à l'autre. Certains langages réclament le nombre d'éléments, d'autre le plus grand indice... C'est donc là aussi une affaire de conventions).

En nous calquant sur les choix les plus fréquents dans les langages de programmations, nous déciderons ici arbitrairement et une bonne fois pour toutes que :

- les "cases" sont numérotées à partir de zéro, autrement dit que le plus petit indice est zéro.
- lors de la déclaration d'un tableau, on précise la plus grande valeur de l'indice (différente, donc, du nombre de cases du tableau, puisque si on veut 12 emplacements, le plus grand indice sera 11). Au début, ça déroute, mais vous verrez, avec le temps, on se fait à tout, même au pire.

```
Tableau Note[11] en Entier
```

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, bien sûr, mais aussi tableaux de caractères, tableaux de booléens, tableaux de tout ce qui existe dans un langage donné comme type de variables. Par contre, hormis dans quelques rares langages, on ne peut pas faire un mixage de types différents de valeurs au sein d'un même tableau.

L'énorme avantage des tableaux, c'est qu'on va pouvoir les traiter en faisant des boucles. Par exemple, pour effectuer notre calcul de moyenne, cela donnera par exemple :

```
Tableau Note[11] en Numérique
Variables Moy, Som en Numérique
Début
Pour i ← 0 à 11
  Ecrire "Entrez la note n°", i
  Lire Note[i]
i Suivant
Som ← 0
Pour i ← 0 à 11
  Som ← Som + Note[i]
i Suivant
Moy ← Som / 12
Fin
```

NB : On a fait deux boucles successives pour plus de lisibilité, mais on aurait tout aussi bien pu n'en écrire qu'une seule dans laquelle on aurait tout fait d'un seul coup.

Remarque générale : l'indice qui sert à désigner les éléments d'un tableau peut être exprimé directement comme un nombre en clair, mais il peut être aussi une variable, ou une expression calculée.

Dans un tableau, la valeur d'un indice doit toujours :

- **être égale au moins à 0** (dans quelques rares langages, le premier élément d'un tableau porte l'indice 1). Mais comme je l'ai déjà écrit plus haut, nous avons choisi ici de commencer la numérotation des indices à zéro, comme c'est le cas en

langage C et en Visual Basic. Donc attention, Truc[6] est le septième élément du tableau Truc !

- **être un nombre entier** Quel que soit le langage, l'élément Truc[3,1416] n'existe jamais.
- **être inférieure ou égale au nombre d'éléments du tableau** (moins 1, si l'on commence la numérotation à zéro). Si le tableau Bidule a été déclaré comme ayant 25 éléments, la présence dans une ligne, sous une forme ou sous une autre, de Bidule[32] déclenchera automatiquement une erreur.

Je le re-re-répète, si l'on est dans un langage où les indices commencent à zéro, il faut en tenir compte à la déclaration :

```
Tableau Note[13] en Numérique
```

...créera un tableau de 14 éléments, le plus petit indice étant 0 et le plus grand 13.

LE GAG DE LA JOURNEE
Il consiste à confondre, dans sa tête et / ou dans un algorithme, l'**indice** d'un élément d'un tableau avec le **contenu** de cet élément. La troisième maison de la rue n'a pas forcément trois habitants, et la vingtième vingt habitants. En notation algorithmique, il n'y a aucun rapport entre i et truc[i].

Exercice 1

Que produit l'algorithme suivant ?

```
Tableau Nb[5] en Entier
Variable i en Entier
Début
Pour i ← 0 à 5
  Nb[i] ← i * i
i suivant
Pour i ← 0 à 5
  Ecrire Nb[i]
i suivant
Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?

Exercice 2

Que produit l'algorithme suivant ?

```
Tableau N[6] en Entier
Variables i, k en Entier
Début
N[0] ← 1
```

```
Pour k ← 1 à 6
  N[k] ← N[k-1] + 2
k Suivant
Pour i ← 0 à 6
  Ecrire N[i]
i suivant
Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?

--- Corrigé 1 :

Cet algorithme remplit un tableau avec six valeurs : 0, 1, 4, 9, 16, 25. Il les écrit ensuite à l'écran. Simplification :

```
Tableau Nb[5] en Numérique
Variable i en Numérique
Début
Pour i ← 0 à 5
  Nb[i] ← i * i
  Ecrire Nb[i]
i Suivant
Fin
```

```
Variable i en Numérique
Début
Pour i ← 0 à 5
  Ecrire i * i
i Suivant
Fin
```

--- Corrigé 2 :

Cet algorithme remplit un tableau avec les sept valeurs : 1, 3, 5, 7, 9, 11, 13. Il les écrit ensuite à l'écran. Simplification :

```
Tableau N[6] en Numérique
Variables i, k en Numérique
Début
N[0] ← 1
Ecrire N[0]
Pour k ← 1 à 6
  N[k] ← N[k-1] + 2
  Ecrire N[k]
```

```
k Suivant
Fin
```

3. Tableaux dynamiques

Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter un tableau. Bien sûr, une solution consisterait à déclarer un tableau gigantesque (10 000 éléments, pourquoi pas, au diable les varices) pour être sûr que « ça rentre ». Mais d'une part, on n'en sera jamais parfaitement sûr, d'autre part, en raison de l'immensité de la place mémoire réservée – et la plupart du temps non utilisée, c'est un gâchis préjudiciable à la rapidité, voire à la viabilité, de notre algorithme.

Aussi, pour parer à ce genre de situation, a-t-on la possibilité de déclarer le tableau sans préciser au départ son nombre d'éléments. Ce n'est que dans un second temps, au cours du programme, que l'on va fixer ce nombre via une instruction de redimensionnement : **Redim**.

Notez que **tant qu'on n'a pas précisé le nombre d'éléments d'un tableau, d'une manière ou d'une autre, ce tableau est inutilisable**.

Exemple : on veut faire saisir des notes pour un calcul de moyenne, mais on ne sait pas combien il y aura de notes à saisir. Le début de l'algorithme sera quelque chose du genre :

```
Tableau Notes[] en Numérique
Variable nb en Numérique
Début
Ecrire "Combien y a-t-il de notes à saisir ?"
Lire nb
Redim Notes[nb-1]
…
```

Cette technique n'a rien de sorcier, mais elle fait partie de l'arsenal de base de la programmation en gestion.

Exercice 1

Ecrivez un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives.

Exercice 2

Ecrivez un algorithme calculant la somme des valeurs d'un tableau (on suppose que le tableau a été préalablement saisi).

Exercice 3

Ecrivez un algorithme constituant un tableau, à partir de deux tableaux de même longueur préalablement saisis. Le nouveau tableau sera la somme des éléments des deux tableaux de départ.

Tableau 1 :

4	8	7	9	1	5	4	6

Tableau 2 :

7	6	5	2	1	3	7	4

Tableau à constituer :

11	14	12	11	2	8	11	10

--- Corrigé 1 :

```
Variables Nb, Nbpos, Nbneg en Numérique
Tableau T[] en Numérique
Variable i en Numérique
Debut
Ecrire "Entrez le nombre de valeurs :"
Lire Nb
Redim T[Nb-1]
Nbpos ← 0
Nbneg ← 0
Pour i ← 0 à Nb - 1
  Ecrire "Entrez le nombre n° ", i + 1
  Lire T[i]
  Si T[i] > 0 alors
    Nbpos ← Nbpos + 1
  Sinon
    Nbneg ← Nbneg + 1
  Finsi
i Suivant
Ecrire "Nombre de valeurs positives : ", Nbpos
Ecrire "Nombre de valeurs négatives : ", Nbneg
Fin
```

--- Corrigé 2 :

```
Variables i, Som, N en Numérique
Tableau T[] en Numérique
Debut
```

… (on ne programme pas la saisie du tableau, dont on suppose qu'il compte N éléments]

```
Redim T[N-1]
…
Som ← 0
Pour i ← 0 à N - 1
  Som ← Som + T[i]
i Suivant
Ecrire "Somme des éléments du tableau : ", Som
Fin
```

--- Corrigé 3 :

```
Variables i, N en Numérique
Tableaux T1[], T2[], T3[] en Numérique
Debut
```

… (on suppose que T1 et T2 comptent N éléments, et qu'ils sont déjà saisis)

```
Redim T3[N-1]
…
Pour i ← 0 à N - 1
  T3[i] ← T1[i] + T2[i]
i Suivant
Fin
```

I-Techniques Rusees

« *Informatique : alliance d'une science inexacte et d'une activité humaine faillible* » - Luc Fayard

Une fois n'est pas coutume, ce chapitre n'a pas pour but de présenter un nouveau type de données, un nouveau jeu d'instructions, ou que sais-je encore.

Son propos est de détailler quelques techniques de programmation qui possèdent deux grands points communs :

- leur connaissance est parfaitement indispensable
- elles sont un rien finaudes

Et que valent quelques kilos d'aspirine, comparés à l'ineffable bonheur procuré par la compréhension suprême des arcanes de l'algorithmique ? Hein ?

1. Tri d'un tableau : le tri par sélection

Première de ces ruses de sioux, et par ailleurs tarte à la crème absolue du programmeur, donc : le tri de tableau.

Combien de fois au cours d'une carrière (brillante) de développeur a-t-on besoin de ranger des valeurs dans un ordre donné ? C'est inimaginable. Aussi, plutôt qu'avoir à réinventer à chaque fois la roue, le fusil à tirer dans les coins, le fil à couper le roquefort et la poudre à maquiller, vaut-il mieux avoir assimilé une ou deux techniques solidement éprouvées, même si elles paraissent un peu ardues au départ.

Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons deux : le tri par sélection, et le tri à bulles. Champagne !

Commençons par le tri par sélection.

Admettons que le but de la manœuvre soit de trier un tableau de 12 éléments dans l'ordre croissant. La technique du tri par sélection est la suivante : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis en met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier. Par exemple, si l'on part de :

45	122	12	3	21	78	64	53	89	28	84	46

On commence par rechercher, parmi les 12 valeurs, quel est le plus petit élément , et où il se trouve. On l'identifie en quatrième position (c'est le nombre 3), et on l'échange alors avec le premier élément (le nombre 45). Le tableau devient ainsi :

3	122	12	**45**	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément, mais cette fois, **seulement à partir du deuxième** (puisque le premier est maintenant correct, on n'y touche plus). On le trouve en troisième position (c'est le nombre 12). On échange donc le deuxième avec le troisième :

3	**12**	**122**	45	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément à partir du troisième (puisque les deux premiers sont maintenant bien placés), et on le place correctement, en l'échangeant, ce qui donnera in fine :

3	12	**21**	45	**122**	78	64	53	89	28	84	46

Et cetera, et cetera, jusqu'à l'avant dernier.

En bon français, nous pourrions décrire le processus de la manière suivante :

- Boucle principale : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.
- Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel et le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

Assez joué, il faut maintenant passer à l'algorithme. Celui-ci s'écrit :

```
boucle principale : le point de départ se décale à chaque tour
Pour i ← 0 à 10
on considère provisoirement que t[i] est le plus petit élément
  posmini ← i
on examine tous les éléments suivants
  Pour j ← i + 1 à 11
    Si t[j] < t[posmini] Alors
       posmini ← j
    Finsi
  j suivant
A cet endroit, on sait maintenant où est le plus petit élément. Il ne reste
plus qu'à effectuer la permutation.
  temp ← t[posmini]
  t[posmini] ← t[i]
  t[i] ← temp
On a placé correctement l'élément numéro i, on passe à présent au suivant.
i suivant
```

Une variante :

On peut imaginer une légère variante à cet algorithme, qui correspond à une très légère simplification. Jusqu'à présent, lorsqu'on cherchait à positionner la case numéro i, on parcourait tout le tableau à partir de la case i+1, et c'est qu'après avoir localisé la valeur la plus petite qu'on procédait à l'échange. Mais, après tout, on pourrait tout aussi bien effectuer cet échange au fur et à mesure, à chaque fois qu'on trouve une valeur plus petite.

On va écrire l'algorithme de cette variante dans un instant, mais auparavant, rien ne saurait nous priver du spectacle assez étonnant de cette variante mise en scène et en musique par une sympathique troupe de danse folklorique hongroise, réquisitionnée pour la bonne cause par une fac d'informatique. Attention, c'est un régal pour les yeux et les neurones, mais ça pique un peu les oreilles quand même :

Revenons (ouf) dans le monde du silence, et écrivons l'algorithme correspondant à cette variante :

```
boucle principale : le point de départ se décale à chaque tour
Pour i ← 0 à 10
on examine tous les éléments suivants
  Pour j ← i + 1 à 11
    Si t[j] < t[i] Alors
      on fait l'échange !
      temp ← t[i]
      t[i] ← t[j]
      t[j] ← temp
    Finsi
  j suivant
On a placé correctement l'élément numéro i, on passe à présent au suivant.
i suivant
```

2. Un exemple de flag : la recherche dans un tableau

Nous allons maintenant nous intéresser au maniement habile d'une variable booléenne : la technique dite du « **flag** ».

Le flag, en anglais, est un petit drapeau, qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas. Et, aussitôt que cet événement a lieu, le petit drapeau se lève (la variable booléenne change de valeur). Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Tout ceci peut vous sembler un peu fumeux, mais cela devrait s'éclairer à l'aide d'un exemple extrêmement fréquent : la recherche de l'occurrence d'une valeur dans un tableau. On en profitera au passage pour corriger une erreur particulièrement fréquente chez le programmeur débutant.

Soit un tableau comportant, disons, 20 valeurs. L'on doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.

La première étape, évidente, consiste à écrire les instructions de lecture / écriture, et la boucle – car il y en a manifestement une – de parcours du tableau :

```
Tableau Tab[19] en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Pour i ← 0 à 19
 ???
i suivant
Fin
```

Il nous reste à combler les points d'interrogation de la boucle Pour. Évidemment, il va falloir comparer N à chaque élément du tableau : si les deux valeurs sont égales, alors bingo, N fait partie du tableau. Cela va se traduire, bien entendu, par un Si … Alors … Sinon. Et voilà le programmeur raisonnant hâtivement qui se vautre en écrivant :

```
Tableau Tab[19] en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Pour i ← 0 à 19
  Si N = Tab[i] Alors
    Ecrire N "fait partie du tableau"
  Sinon
    Ecrire N "ne fait pas partie du tableau"
  FinSi
i suivant
Fin
```

Et patatras, cet algorithme est une véritable catastrophe.

Il suffit d'ailleurs de le faire tourner mentalement pour s'en rendre compte. De deux choses l'une : ou bien la valeur N figure dans le tableau, ou bien elle n'y figure pas. Mais dans tous les cas, **l'algorithme ne doit produire qu'une seule réponse, quel que soit le nombre d'éléments que compte le tableau. Or, l'algorithme ci-dessus envoie à l'écran autant de messages qu'il y a de valeurs dans le tableau**, en l'occurrence pas moins de 20 !

Il y a donc une erreur manifeste de conception : l'écriture du message ne peut se trouver à l'intérieur de la boucle : elle doit figurer à l'extérieur. On sait si la valeur était dans le tableau ou non **uniquement lorsque le balayage du tableau est entièrement accompli**.

Nous réécrivons donc cet algorithme en plaçant le test après la boucle. Faute de mieux, on se contentera de faire dépendre pour le moment la réponse d'une variable booléenne que nous appellerons Trouvé.

```
Tableau Tab[19] en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Pour i ← 0 à 19
    ???
i suivant
Si Trouvé Alors
  Ecrire "N fait partie du tableau"
Sinon
  Ecrire "N ne fait pas partie du tableau"
FinSi
Fin
```

Il ne nous reste plus qu'à gérer la variable Trouvé. Ceci se fait en deux étapes.

- un test figurant dans la boucle, indiquant lorsque la variable Trouvé doit devenir vraie (à savoir, lorsque la valeur N est rencontrée dans le tableau). Et attention : le test est asymétrique. Il ne comporte pas de "sinon". On reviendra là dessus dans un instant.
- last, but not least, l'affectation par défaut de la variable Trouvé, dont la valeur de départ doit être évidemment Faux.

Au total, l'algorithme complet – et juste ! – donne :

```
Tableau Tab[19] en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Trouvé ← Faux
Pour i ← 0 à 19
  Si N = Tab[i] Alors
    Trouvé ← Vrai
  FinSi
i suivant
Si Trouvé Alors
  Ecrire "N fait partie du tableau"
Sinon
  Ecrire "N ne fait pas partie du tableau"
FinSi
Fin
```

Méditons un peu sur cette affaire.

La difficulté est de comprendre que dans une recherche, le problème ne se formule pas de la même manière selon qu'on le prend par un bout ou par un autre. On peut résumer l'affaire ainsi : **il suffit que N soit égal à une seule valeur de Tab pour qu'elle fasse partie du tableau. En revanche, il faut qu'elle soit différente de toutes les valeurs de Tab pour qu'elle n'en fasse pas partie**.

Voilà la raison qui nous oblige à passer par une variable booléenne **, un « drapeau » qui peut se lever, mais jamais se rabaisser**. Et cette technique de flag (que nous pourrions élégamment surnommer « gestion asymétrique de variable booléenne ») doit être mise en œuvre chaque fois que l'on se trouve devant pareille situation.

Autrement dit, connaître ce type de raisonnement est indispensable, et savoir le reproduire à bon escient ne l'est pas moins.

Dans ce cas précis, il est vrai, on pourrait à juste titre faire remarquer que l'utilisation de la technique du flag, si elle permet une subtile mais ferme progression pédagogique, ne donne néanmoins pas un résultat optimum. En effet, dans l'hypothèse où la machine trouve la valeur recherchée quelque part au milieu du tableau, notre algorithme continue – assez bêtement, il faut bien le dire – la recherche jusqu'au bout du tableau, alors qu'on pourrait s'arrêter net.

Le meilleur algorithme possible, même s'il n'utilise pas de flag, consiste donc à remplacer la boucle **Pour** par une boucle **TantQue** : en effet, là, on ne sait plus combien de tours de boucle il va falloir faire (puisqu'on risque de s'arrêter avant la fin du tableau). Pour savoir quelle condition suit le TantQue, raisonnons à l'envers : on s'arrêtera quand on aura trouvé la

valeur cherchée... ou qu'on sera arrivés à la fin du tableau. Appliquons la transformation de Morgan : Il faut donc poursuivre la recherche tant qu'on n'a pas trouvé la valeur et qu'on n'est pas parvenu à la fin du tableau. Démonstration :

```
Tableau Tab[19] en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
i ← 0
TantQue N <> T[i] et i < 19
   i ← i + 1
FinTantQue
Si N = Tab[i] Alors
   Ecrire "N fait partie du tableau"
Sinon
   Ecrire "N ne fait pas partie du tableau"
FinSi
Fin
```

3. Tri de tableau + flag = tri à bulles

Et maintenant, nous en arrivons à la formule magique : tri de tableau + flag = tri à bulles.

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel **tout élément est plus petit que celui qui le suit**. Cette constatation percutante semble digne de M. de Lapalisse, un ancien voisin à moi. Mais elle est plus profonde – et plus utile - qu'elle n'en a l'air.

En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands « remontent » ainsi peu à peu vers les dernières places, ce qui explique la charmante dénomination de « tri à bulle ». Comme quoi l'algorithmique n'exclut pas un minimum syndical de sens poétique.

Et à l'appui de cette dernière affirmation, s'il en était besoin, nous retrouvons les joyeux drilles transylvaniens, qui sur un rythme endiablé, nous font cette fois une démonstration de tri à bulles.

Toujours aussi bluffant, hein ?

Vous pouvez à présent retirer vos boules quiès et passer à l'écriture de l'agorithme. En quoi le tri à bulles implique-t-il l'utilisation d'un flag ? Eh bien, par ce qu'on ne sait jamais par avance combien de remontées de bulles on doit effectuer. En fait, tout ce qu'on peut dire, c'est qu'on devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés. Ceci est typiquement un cas de question « asymétrique » : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié.

Nous baptiserons le flag Yapermute, car cette variable booléenne va nous indiquer si nous avons ou non procédé à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est signe que le tableau est trié, et donc qu'on peut arrêter la machine à bulles). La boucle principale sera alors :

```
Variable Yapermute en Booléen
Début
…
TantQue Yapermute
…
FinTantQue
Fin
```

Que va-t-on faire à l'intérieur de la boucle ? Prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, et procéder à un échange si nécessaire. C'est parti :

```
Variable Yapermute en Booléen
Début
…
TantQue Yapermute
Pour i ← 0 à 10
  Si t[i] > t[i+1] Alors
    temp ← t[i]
    t[i] ← t[i+1]
    t[i+1] ← temp
  Finsi
i suivant
Fin
```

Mais il ne faut pas oublier un détail capital : la gestion de notre flag. L'idée, c'est que cette variable va nous signaler le fait qu'il y a eu au moins une permutation effectuée. Il faut donc :

- lui attribuer la valeur Vrai dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
- la remettre à Faux à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés),
- dernier point, il ne faut pas oublier de lancer la boucle principale, et pour cela de donner la valeur Vrai au flag au tout départ de l'algorithme.

La solution complète donne donc :

```
Variable Yapermute en Booléen
Début
…
Yapermut ← Vrai
TantQue Yapermut
  Yapermut ← Faux
  Pour i ← 0 à 10
    Si t[i] > t[i+1] alors
```

```
      temp ← t[i]
      t[i] ← t[i+1]
      t[i+1] ← temp
      Yapermut ← Vrai
    Finsi
  i suivant
FinTantQue
Fin
```

Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie de l'arsenal du programmeur bien armé.

4. La recherche dichotomique

Je ne sais pas si on progresse vraiment en algorithmique, mais en tout cas, qu'est-ce qu'on apprend comme vocabulaire !

Blague dans le coin, nous allons terminer ce chapitre migraineux par une technique célèbre de recherche, qui révèle toute son utilité lorsque le nombre d'éléments est très élevé. Par exemple, imaginons que nous ayons un programme qui doive vérifier si un mot existe dans le dictionnaire. Nous pouvons supposer que le dictionnaire a été préalablement entré dans un tableau (à raison d'un mot par emplacement). Ceci peut nous mener à, disons à la louche, 40 000 mots.

Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier. Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20 000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, un être humain qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots sont triés.

Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve pile poil au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans le première moitié du dico. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié.

A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne en terme de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.

- Au départ, on cherche le mot parmi 40 000.
- Après le test n°1, on ne le cherche plus que parmi 20 000.
- Après le test n°2, on ne le cherche plus que parmi 10 000.
- Après le test n°3, on ne le cherche plus que parmi 5 000.
- etc.
- Après le test n°15, on ne le cherche plus que parmi 2.
- Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas. Moralité : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment ! Il n'y a pas photo sur l'écart de performances entre la technique barbare et la technique futée. Attention, toutefois, même si c'est évident, je le répète avec force : la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.

Eh bien maintenant qu'on a vu la méthode, il n'y a plus qu'à traduire en pseudo-code !

```
On suppose qu'on dispose d'un tableau T[N-1] et qu'on y recherche une
valeur numérique X
On déclare trois numériques (début, milieu et fin) et un booléen (Trouvé)
...
Début
…
Trouvé ← Faux
Debut ← 0
Fin ← N-1
TantQue Non Trouvé ET Debut <= Fin
  milieu ← (début + fin)/2
  Si T[milieu]=X Alors
    Trouvé ← Vrai
  SinonSi T[milieu] < X Alors
    Debut ← milieu + 1
  Sinon
    Fin ← milieu - 1
  FinSi
FinTantQue
À l'issue de la boucle, la variable Trouvé contient le résultat
...
Fin
```

Au risque de me répéter, la compréhension et la maîtrise du principe du flag, du tri, et de la recherche (dichotomique ou non) font partie du bagage indispensable du programmeur bien outillé.

Exercice 1

Ecrivez un algorithme qui permette de saisir un nombre quelconque de valeurs, et qui les range au fur et à mesure dans un tableau. Le programme, une fois la saisie terminée, doit dire si les éléments du tableau sont tous consécutifs ou non.

Par exemple, si le tableau est :

12	13	14	15	16	17	18

ses éléments sont tous consécutifs. En revanche, si le tableau est :

9	10	11	15	16	17	18

ses éléments ne sont pas tous consécutifs.

Exercice 2

Ecrivez un algorithme qui trie un tableau dans l'ordre décroissant.

Vous écrirez bien entendu deux versions de cet algorithme, l'une employant le tri par sélection, l'autre le tri à bulles.

Exercice 3

Ecrivez un algorithme qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement saisi (« les premiers seront les derniers… »)

--- corrigé 1 :

```
Variables Nb, i en Entier
Variable Flag en Booleen
Tableau T[] en Entier
Debut
Ecrire "Entrez le nombre de valeurs :"
Lire Nb
Redim T[Nb-1]
Pour i ← 0 à Nb - 1
  Ecrire "Entrez le nombre n° ", i + 1
  Lire T[i]
i Suivant
Flag ← Vrai
Pour i ← 1 à Nb - 1
  Si T[i] <> T[i - 1] + 1 Alors
    Flag ← Faux
  FinSi
i Suivant
Si Flag Alors
  Ecrire "Les nombres sont consécutifs"
```

```
Sinon
  Ecrire "Les nombres ne sont pas consécutifs"
FinSi
Fin
```

Cette programmation est sans doute la plus spontanée, mais elle présente le défaut d'examiner la totalité du tableau, même lorsqu'on découvre dès le départ deux éléments non consécutifs. Aussi, dans le cas d'un grand tableau, est-elle dispendieuse en temps de traitement. Une autre manière de procéder serait de sortir de la boucle dès que deux éléments non consécutifs sont détectés. La deuxième partie de l'algorithme deviendrait donc :

```
i ← 1
TantQue T[i] = T[i – 1] + 1 et i < Nb - 1
  i ← i + 1
FinTantQue
Si T[i] = T[i – 1] + 1 Alors
  Ecrire "Les nombres sont consécutifs"
Sinon
  Ecrire "Les nombres ne sont pas consécutifs"
FinSi
```

--- corrigé 2 :

On suppose que N est le nombre d'éléments du tableau. Tri par insertion :

```
…
Pour i ← 0 à N - 2
  posmaxi = i
  Pour j ← i + 1 à N - 1
    Si t[j] > t[posmaxi] alors
      posmaxi ← j
    Finsi
  j suivant
  temp ← t[posmaxi]
  t[posmaxi] ← t[i]
  t[i] ← temp
i suivant
Fin
```

Tri à bulles :

```
…
Yapermut ← Vrai
TantQue Yapermut
  Yapermut ← Faux
  Pour i ← 0 à N - 2
    Si t[i] < t[i + 1] Alors
```

```
      temp ← t[i]
      t[i] ← t[i + 1]
      t[i + 1] ← temp
      Yapermut ← Vrai
    Finsi
  i suivant
FinTantQue
Fin
```

--- corrigé 3 :

On suppose que n est le nombre d'éléments du tableau préalablement saisi

```
…
Pour i ← 0 à (N-1)/2
  Temp ← T[i]
  T[i] ← T[N-1-i]
  T[N-1-i] ← Temp
i suivant
Fin
```

II-Tableaux Multidimensionnels

« *Le vrai problème n'est pas de savoir si les machines pensent, mais de savoir si les hommes pensent* » - B.F. Skinner

« *La question de savoir si un ordinateur peut penser n'est pas plus intéressante que celle de savoir si un sous-marin peut nager* » - Edgar W. Dijkstra

Ceci n'est pas un dérèglement de votre téléviseur. Nous contrôlons tout, nous savons tout, et les phénomènes paranormaux que vous constatez sont dus au fait que vous êtes passés dans… la quatrième dimension (musique angoissante : « tintintin… »).

Oui, enfin bon, avant d'attaquer la quatrième, on va déjà se coltiner la deuxième.

1. Pourquoi plusieurs dimensions ?

Une seule ne suffisait-elle pas déjà amplement à notre bonheur, me demanderez-vous ? Certes, répondrai-je, mais vous allez voir qu'avec deux (et davantage encore) c'est carrément le nirvana.

Prenons le cas de la modélisation d'un jeu de dames, et du déplacement des pions sur le damier. Je rappelle qu'un pion qui est sur une case blanche peut se déplacer (pour simplifier) sur les quatre cases blanches adjacentes.

Avec les outils que nous avons abordés jusque là, le plus simple serait évidemment de modéliser le damier sous la forme d'un tableau. Chaque case est un emplacement du tableau, qui contient par exemple 0 si elle est vide, et 1 s'il y a un pion. On attribue comme indices aux cases les numéros 1 à 8 pour la première ligne, 9 à 16 pour la deuxième ligne, et ainsi de suite jusqu'à 64.

Arrivés à ce stade, les fines mouches du genre de Cyprien L. m'écriront pour faire remarquer qu'un damier, cela possède 100 cases et non 64, et qu'entre les damiers et les échiquiers, je me suis joyeusement emmêlé les pédales. A ces fines mouches, je ferai une double réponse de prof :

1. C'était pour voir si vous suiviez.

2. Si le prof décide contre toute évidence que les damiers font 64 cases, c'est le prof qui a raison et l'évidence qui a tort. Rompez.

Reprenons. Un pion placé dans la case numéro i, autrement dit la valeur 1 de Cases[i], peut bouger vers les cases contiguës en diagonale. Cela va nous obliger à de petites acrobaties intellectuelles : la case située juste au-dessus de la case numéro i ayant comme indice i-8, les cases valables sont celles d'indice i-7 et i-9. De même, la case située juste en dessous ayant comme indice i+8, les cases valables sont celles d'indice i+7 et i+9.

Bien sûr, on peut fabriquer tout un programme comme cela, mais le moins qu'on puisse dire est que cela ne facilite pas la clarté de l'algorithme.

Il serait évidemment plus simple de modéliser un damier par… un damier !

2. Tableaux à deux dimensions

L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par une seule, mais par **deux coordonnées**.

Un tel tableau se déclare ainsi :

```
Tableau Cases[7, 7] en Numérique
```

Cela veut dire : réserve moi un espace de mémoire pour 8 x 8 entiers, et quand j'aurai besoin de l'une de ces valeurs, je les repèrerai par deux indices (comme à la bataille navale, ou Excel, la seule différence étant que pour les coordonnées, on n'utilise pas de lettres, juste des chiffres).

Pour notre problème de dames, les choses vont sérieusement s'éclaircir. La case qui contient le pion est dorénavant Cases[i, j]. Et les quatre cases disponibles sont Cases[i-1, j-1], Cases[i-1, j+1], Cases[i+1, j-1] et Cases[i+1, j+1].

REMARQUE ESSENTIELLE :
Il n'y a aucune différence qualitative entre un tableau à deux dimensions [i, j] et un tableau à une dimension [i * j]. De même que le jeu de dames qu'on vient d'évoquer, tout problème qui peut être modélisé d'une manière peut aussi être modélisé de l'autre. Simplement, l'une ou l'autre de ces techniques correspond plus spontanément à tel ou tel problème, et facilite donc (ou complique, si on a choisi la mauvaise option) l'écriture et la lisibilité de l'algorithme.

Une autre remarque : une question classique à propos des tableaux à deux dimensions est de savoir si le premier indice représente les lignes ou le deuxième les colonnes, ou l'inverse. Je ne répondrai pas à cette question non parce que j'ai décidé de bouder, mais **parce qu'elle n'a aucun sens**. « Lignes » et « Colonnes » sont des concepts graphiques, visuels, qui s'appliquent à des objets du monde réel ; les indices des tableaux ne sont que des coordonnées logiques, pointant sur des adresses de mémoire vive. Si cela ne vous convainc pas, pensez à un jeu de bataille navale classique : les lettres doivent-elles désigner les lignes et les chiffres les colonnes ? Aucune importance ! Chaque joueur peut même choisir une convention différente, aucune importance ! L'essentiel est qu'une fois une convention choisie, un joueur conserve la même tout au long de la partie, bien entendu.

Exercice 1

Écrivez un algorithme remplissant un tableau de 6 sur 13, avec des zéros.

Exercice 2

Quel résultat produira cet algorithme ?

```
Tableau X[1, 2] en Entier
Variables i, j, val en Entier
Début
Val ← 1
Pour i ← 0 à 1
  Pour j ← 0 à 2
    X[i, j] ← Val
```

```
    Val ← Val + 1
  j Suivant
i Suivant
Pour i ← 0 à 1
  Pour j ← 0 à 2
    Ecrire X[i, j]
  j Suivant
i Suivant
Fin
```

Exercice 3

Soit un tableau T à deux dimensions [12, 8] préalablement rempli de valeurs numériques.

Écrire un algorithme qui recherche la plus grande valeur au sein de ce tableau.

--- corrigé 1 :

```
Tableau Truc[5, 12] en Entier
Debut
Pour i ← 0 à 5
  Pour j ← 0 à 12
    Truc[i, j] ← 0
  j Suivant
i Suivant
Fin
```

--- corrigé 2 :

Cet algorithme remplit un tableau de la manière suivante:

```
X[0, 0] = 1
X[0, 1] = 2
X[0, 2] = 3
X[1, 0] = 4
X[1, 1] = 5
X[1, 2] = 6
```

Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

--- corrigé 3 :

```
Variables i, j, iMax, jMax en Numérique
Tableau T[12, 8] en Numérique
```

Le principe de la recherche dans un tableau à deux dimensions est strictement le même que dans un tableau à une dimension, ce qui ne doit pas nous étonner. La seule chose qui change, c'est qu'ici le balayage requiert deux boucles imbriquées, au lieu d'une seule.

```
Debut
...
iMax ← 0
jMax ← 0
Pour i ← 0 à 12
  Pour j ← 0 à 8
    Si T[i,j] > T[iMax,jMax] Alors
      iMax ← i
      jMax ← j
    FinSi
  j Suivant
i Suivant
Ecrire "Le plus grand élément est ", T[iMax, jMax]
Ecrire "Il se trouve aux indices ", iMax, "; ", jMax
Fin
```

3. Tableaux à n dimensions

Si vous avez compris le principe des tableaux à deux dimensions, sur le fond, il n'y a aucun problème à passer au maniement de tableaux à trois, quatre, ou pourquoi pas neuf dimensions. C'est exactement la même chose. Si je déclare un tableau Titi[2, 4, 3, 3], il s'agit d'un espace mémoire contenant 3 x 5 x 4 x 4 = 240 valeurs. Chaque valeur y est repérée par quatre coordonnées.

Le principal obstacle au maniement systématique de ces tableaux à plus de trois dimensions est que le programmeur, quand il conçoit son algorithme, aime bien faire des petits gribouillis, des dessins immondes, imaginer les boucles dans sa tête, etc. Or, autant il est facile d'imaginer concrètement un tableau à une dimension, autant cela reste faisable pour deux dimensions, autant cela devient l'apanage d'une minorité privilégiée pour les tableaux à trois dimensions (je n'en fais malheureusement pas partie) et hors de portée de tout mortel au-delà. C'est comme ça, l'esprit humain a du mal à se représenter les choses dans l'espace, et crie grâce dès qu'il saute dans l'hyperespace (oui, c'est comme ça que ça s'appelle au delà de trois dimensions).

Donc, pour des raisons uniquement pratiques, les tableaux à plus de trois dimensions sont rarement utilisés par des programmeurs non matheux (car les matheux, de par leur formation, ont une fâcheuse propension à manier des espaces à n dimensions comme qui rigole, mais ce sont bien les seuls, et laissons les dans leur coin, c'est pas des gens comme nous).

CHAP 2. CALCULEZ LA COMPLEXITE ALGORITHMIQUE

Vous avez pu voir qu'un algorithme peut être créé de mille et une façons. Mais comment pouvez-vous savoir si votre algorithme est plus efficace qu'un autre, ou même savoir s'il est performant ?

Pour cela, vous allez devoir analyser la complexité de votre algorithme. Ainsi, vous pourrez conclure que cette complexité est convenable, ou la comparer avec d'autres algorithmes.

I-Qu'est-ce que la complexité algorithmique ?

La complexité algorithmique est un concept très important qui permet de **comparer les algorithmes** afin de trouver celui qui est le plus efficace. Il existe une notation standard qui s'appelle **big O** et qui permet de mesurer la performance d'un algorithme. Vous verrez, au fur et à mesure des explications, la méthode de **calcul**.

Avant d'entrer dans le détail de son calcul, laissez-moi vous conter une petite histoire.

II-La complexité exponentielle

Nous sommes dans le Midi, au cœur d'une belle forêt, et nous nous promenons paisiblement. Soudain, un éclat de lumière attire votre regard. Vous vous approchez. Quelle n'est pas votre surprise lorsque vous apercevez un coffre qui semble être sorti tout droit d'un bateau pirate !

Vous soulevez le coffre et tombez nez à nez avec un **cadenas à trois chiffres**, en fonte, bien décidé à ne pas vous laisser accéder au trésor tant escompté. Quand bien même ! Nous décidons de relever le défi et de tester rapidement, une à une, toutes les combinaisons.

Plutôt que de les tester dans le désordre, car il serait impossible de se souvenir de toutes les tentatives, nous réfléchissons à **différentes stratégies**. L'une d'elles consiste à **essayer le nombre le plus petit (000) puis le nombre suivant (001)** et ainsi de suite jusqu'à atteindre 999. En effet, nous calculons que cette stratégie nous prendra, dans le pire des cas, 30 minutes.

Effectivement, quelques minutes plus tard, le cadenas s'ouvre. Hourra ! Le code était 123. Facile ! Notre stratégie était efficace et nous nous félicitons d'être si intelligents.

Voyons maintenant comment décrire ce problème avec le pseudo-code :

```
Algorithme cadenas_3_chiffres
Début
    Pour i allant de 0 jusqu'à 9 :
        Pour j allant de 0 jusqu'à 9 :
            Pour k allant de 0 jusqu'à 9 :
                Si code == combinaison :
                    Arrêter les boucles
                Fin Si
            Fin Pour
        Fin Pour
    Fin Pour
Fin
```

Nous avons dans cet algorithme 3 boucles allant de 0 à 9 donc 10 x 10 x 10 itérations, soit **10^3 itérations**. Avec la notation **Big O**, on dira que la complexité temporelle de cet algorithme est de $O(10^3)$.

Nous ouvrons le coffre... pour en découvrir un second, plus petit, comportant un **cadenas à 4 chiffres**. Tristesse.

Forts de notre premier succès, **nous nous basons sur le même algorithme pour trouver le code**. Au bout d'une heure et demie, las, nous abandonnons l'idée. C'est le plus raisonnable : tester toutes les combinaisons possibles aurait pris plus de 5 heures.

Nous reprenons notre marche, le petit coffre sous le bras, et partons dans un grand débat : comment en est-on arrivés là ? Comment le temps de calcul peut-il passer de 30 minutes à 5 heures en ajoutant un simple chiffre ? **Le secret, c'est la complexité de notre algorithme**.

Au risque de ne pas vous surprendre, plus il y a de chiffres dans le code, plus il sera long à trouver. **Mais plus long comment ?**

Si le code a **3 chiffres**, il faut tester **1 000 combinaisons** (eh oui, tous les nombres entre 000 et 999). En revanche, s'il en a **4**, il faut en tester **10 000**. S'il en avait eu **5**, il aurait fallu en tester **100 000**, et ainsi de suite. Nous nous apercevons donc que **notre algorithme d'ouverture de coffre dépend du nombre de chiffres du code**.

Plus précisément, le temps mis à l'ouvrir est multiplié par 10 à chaque fois que l'on ajoute un chiffre : 10 pour un code à 1 chiffre, 100 (10 x 10) pour un code à 2 chiffres, 1 000 (10 x 10 x 10) pour un code à trois chiffres, et ainsi de suite.

Si l'on note **n** le nombre de chiffres de notre cadenas, le temps de calcul est donc 10^n ; ainsi, à l'aide de la notion **big O**, nous pouvons noter que la complexité temporelle de notre algorithme est de **O(** 10^n**)**.

On dit alors que **la complexité est exponentielle.** Très vite, nous nous sommes rendu compte que notre algorithme était impossible à réaliser, car il devenait trop long.

III-La complexité linéaire

Après avoir disserté sur ce sujet, nous avons soudain une idée : et si nous arrêtions de calculer et faisions appel à Bill, notre cousin, celui qui trempe dans des affaires louches ? Il nous a raconté un jour qu'il savait "*écouter les cadenas*". Sa technique est simple : **il tourne la molette du premier chiffre jusqu'à entendre un "*clic*"**. Il sait alors que le chiffre est bon et passe au suivant. Il peut donc trouver les bons chiffres un par un, sans avoir à se soucier des autres.

Évidemment, le jour où il vous a exposé sa théorie vous avez bien ri. À présent, vous êtes dubitatif : vous avez passé deux heures à essayer d'ouvrir des coffres de pirates, alors bon, pourquoi ne pas tenter ?

Au bout de deux minutes, le cadenas à 4 chiffres est ouvert. Youpi ! À présent, nous ouvrons le coffre et en découvrons le contenu. Il est à la hauteur de nos efforts !

Mais comment peut-on aller si vite, alors que nous avons exactement le même nombre de chiffres sur le cadenas ?
C'est très simple ! Afin de trouver le premier chiffre du code, Bill tente 10 combinaisons. Quand il l'a trouvé, il passe au suivant et teste de nouveau 10 combinaisons. Et ainsi de suite pour chaque chiffre !

Il n'aura donc à tester que **40 combinaisons** (10 + 10 + 10 + 10, soit 10 x 4) pour ce cadenas à quatre chiffres (ce qui est mieux que les 10 000 combinaisons que nous nous apprêtions à essayer…).

Voyons maintenant comment décrire cette nouvelle solution avec le pseudo-code :

```
Algorithme cadenas_4_chiffres
Variable
    chiffre_1 ← 0 : ENTIER
    chiffre_2 ← 0 : ENTIER
    chiffre_3 ← 0 : ENTIER
    chiffre_4 ← 0 : ENTIER
Début
    Pour i allant de 0 jusqu'à 9 :
        Si entendre un "clic" :
            chiffre_1 ← i
```

```
        Fin Si
    Fin Pour

    Pour j allant de 0 jusqu'à 9 :
        Si entendre un "clic" :
            chiffre_2 ← j
        Fin Si
    Fin Pour

    Pour k allant de 0 jusqu'à 9 :
        Si entendre un "clic" :
            chiffre_3 ← k
        Fin Si
    Fin Pour

    Pour m allant de 0 jusqu'à 9 :
        Si entendre un "clic" :
            chiffre_4 ← m
        Fin Si
    Fin Pour
Fin
```

Si l'on y regarde de plus près, Bill teste **10 nouvelles combinaisons pour chaque nouveau chiffre du cadenas**. Autrement dit, si n est le nombre de chiffres, il teste 10 x n combinaisons. Ainsi avec la notation **big O**, nous pouvons dire que cet algorithme a une complexité de $O(10xn)$. La "complexité" de son algorithme était bien meilleure que la nôtre.
Nous appelons cela une **complexité linéaire**. Pourquoi ? Car elle augmente **proportionnellement** au nombre de chiffres.

IV-La complexité en temps constant

Tout contents, nous allons rendre visite à Bill pour lui montrer le cadenas et crâner un peu. Pour sûr, il sera impressionné !

Bill vous adresse un regard circonspect. *"Vous voulez vous amuser ? Voilà de quoi faire !"*, vous dit-il, sortant d'un placard un cadenas à **500 chiffres** qui manque de faire écrouler la table de la salle à manger. *"Il faut tester **5 000 combinaisons**, ce qui prendra environ 3 heures. Enfin... Si vous êtes rapides."*

Mince, nous nous retrouvons dans le même problème que tout à l'heure. Aussi efficace que soit sa technique, le nombre de chiffres du cadenas est trop important. Autrement dit, **la taille des données du problème excède la capacité de son algorithme**.

C’est là que passe Jack, votre neveu. Il rigole, **prend une pierre et tape sur le cadenas**. Paf ! Ce dernier s’ouvre d’un coup sec. Effectivement... Cela paraît si simple, maintenant que l’on y pense…

Si l’on s’y attarde de plus près, son algorithme est sacrément efficace ! **Quel que soit le nombre de chiffres, il prend toujours le même temps**. Balèze !

Nous parlons alors de **complexité en temps constant**. On note ce type de complexité avec la notation big O, $O(1)$.

V-La complexité temporelle

Si nous comparons nos différents algorithmes, nous nous rendons compte que nous avons surtout pris en compte le facteur **temps** : notre première solution nous a pris 30 minutes quand celle de Jack ne demande qu’une seconde.

Il en sera de même pour chaque algorithme, que ce soit le tri d’une liste, la décomposition en facteurs premiers ou encore la recherche du cours le plus suivi sur Coursera. Nous parlons alors de **complexité temporelle**.

Quel est l’intérêt, me direz-vous ? Cela nous permet de savoir à l’avance si un algorithme ne se terminera jamais. Bien pratique !

Si nous avions utilisé notre algorithme naïf pour ouvrir le cadenas à 500 chiffres, un milliard d’années ne nous auraient pas suffi à trouver la bonne combinaison. Inutile de dire que nous serions morts, réincarnés et re-morts avant de pouvoir profiter du contenu derrière le cadenas.
Internet nous offre une autre application très importante de ce concept. Imaginons un site de restauration à emporter. Que se passe-t-il lorsque vous cherchez tous les restaurants asiatiques à 1 km de chez vous, ouverts jusqu’à 23 heures et qui livrent à domicile ? L’algorithme va tout faire pour trouver le résultat de votre recherche. Malgré cela, vous n’avez pas envie d’attendre 3 heures. Vous souhaitez que le site affiche la page de résultats en une seconde ou deux, pas plus. Il est donc primordial de trouver l’algorithme le plus efficace qui soit.

VI-La complexité spatiale

Le dernier point à connaître concerne le stockage des données. Lorsque nous réalisons un algorithme en informatique, les informations sont stockées sur la mémoire de l’ordinateur. Or, vous l’aurez deviné, cette mémoire n’est pas infinie. Si nous ne faisons pas attention, un algorithme peut vite occuper tout l’espace libre d’un ordinateur, et le faire planter.

On parle ici de **complexité spatiale** (en espace). Les notations **big O** sont exactement les mêmes que pour la complexité temporelle.

Imaginez un algorithme où l'utilisateur doive entrer n éléments et les enregistrer dans un tableau. Dans ce cas, **la complexité spatiale** se notera avec la notation big O, $O(n)$. Car le tableau contiendra finalement **n** éléments. Et si à l'aide des mêmes n éléments, l'algorithme construit deux tableaux de n éléments, cette fois la complexité spatiale sera de $O(2n)$, car l'algorithme crée 2 tableaux de n cases.

À vous de jouer

Contexte

Tout au long de ce cours, vous avez pu créer des algorithmes pour le labyrinthe. Nous allons en prendre quelques-uns et calculer la complexité temporelle et spatiale de ceux-ci.

Consigne

Nous allons calculé la complexité temporelle et spatiale de 3 algorithmes.

- Nous avons tout d'abord l'algorithme de déplacement suivant :

```
Algorithme déplacement(position_x , position_y)
Variable
    sens ← “” : CHAÎNE DE CARACTÈRES
Début
    sens ← entrer()
    Si sens == “Haut” OU sens == “Bas” OU sens == “Droite” OU sens == “Gauche” :
        Si sens == “Haut” :
            position_y = position_y + 1
        Fin Si
        Si sens == “Bas” :
            position_y = position_y - 1
        Fin Si
        Si sens == “Droite” :
            position_x = position_x + 1
        Fin Si
        Si sens == “Gauche” :
            position_x = position_x - 1
        Fin Si
    Sinon
        afficher “L’entrée n’est pas correcte”
    Fin Si
Fin
```

- Nous avons ensuite l'algorithme suivant qui permet de ramasser un nombre n de clés dans le labyrinthe :

```
Algorithme ramasser
Variable
    joueur_position_x ← 0 : ENTIER
    joueur_position_y ← 0 : ENTIER
    arrivée_position_x ← 5 : ENTIER
    arrivée_position_y ← 5 : ENTIER
    clés[n] : TABLEAU CHAÎNE DE CARACTÈRES
    déplacement ← 0 : ENTIER
    max_déplacement ← 30n : ENTIER
Début
    Tant Que joueur_position_x != arrivée_position_x ET joueur_position_y != arrivée_position_y ET
clés.taille() != n ET déplacement < max_déplacement :
        déplacement(joueur_position_x, joueur_position_y)
        ramasser(clés, joueur_position_x, joueur_position_y)
    Fin Tant Que
Fin
```

- Pour terminer, nous avons l'algorithme de tri suivant :

```
Algorithme Tri(tableau)
    taille ← Taille du tableau
    Pour i allant de 0 jusqu'à taille - 1 :
        Pour j allant de 0 jusqu'à taille - 1 - 1 :
            Si tableau[j+1] < tableau[j] :
                echanger(tableau[j+1], tableau[j])
            Fin Si
        Fin Pour
    Fin Pour
Fin
```

Votre objectif :

- Mesurer la complexité temporelle et spatiale des 3 algorithmes :
    - déplacement ;
    - ramassage ;
    - tri.

Vérifiez votre travail

Voici le résultat à obtenir à l'issue de l'exercice :

- La fonction `déplacement` :
    - complexité temporelle : $O(1)$.
      La complexité est constante, il n'y a aucune boucle ni aucune fonction récursive.
    - complexité spatiale : $O(1)$.
      La mémoire n'est pas affectée par cet algorithme.

- La fonction `ramasser` :
  - complexité temporelle : $O(30n)$.
    Au pire des cas, la boucle va itérer 30 x n fois.
  - complexité spatiale : $O(n)$.
    La complexité est linéaire, le tableau aura au maximum n cases.
- La fonction `tri` :
  - complexité temporelle : $O(n^2)$.
    Il y a deux boucles imbriquées qui itèrent n fois, donc nous avons au maximum n x n itérations.
  - complexité spatiale : $O(1)$.
    La mémoire n'est pas affectée par cet algorithme, car la taille du tableau ne change pas.

En résumé

- La complexité d'un algorithme est une mesure de la quantité de temps et/ou d'espace requise par un algorithme.
- La complexité temporelle est le temps nécessaire à l'exécution d'un algorithme, en fonction de la longueur des données en entrée.
- La complexité spatiale mesure la quantité totale de mémoire dont un algorithme ou une opération a besoin pour s'exécuter en fonction de la longueur de données en entrée.
- La notation big O est une notation standard pour décrire la complexité d'un algorithme.

C'est bientôt terminé, il ne reste plus qu'un chapitre. Je vous propose de voir un concept assez avancé de l'informatique, qu'on appelle la récursivité. Je vous attends avec impatience dans ce dernier chapitre.

CHAP 3. VOYEZ LE MONDE AUTREMENT AVEC LA RECURSIVITE

En programmation, nous avons vu que nous pouvions utiliser des boucles pour répéter une opération. Nous les appelons des boucles *itératives*. Laissez-moi vous présenter un nouveau concept : la récursivité.

Il s'agit d'un concept un peu spécial. Avez-vous déjà rêvé que vous rêviez ? Vous y êtes ! Nous allons parler de concepts qui s'appellent eux-mêmes. C'est parti !

Qu'est-ce que la récursivité ?

La récursivité est un concept qui fait référence à lui-même dans son fonctionnement.

Les poupées russes sont une excellente métaphore de la récursivité. Si vous n'avez jamais joué avec des poupées russes, elles sont exactement ce à quoi elles ressemblent. Chaque poupée est la même sauf pour sa taille ; elles s'ouvrent chacune, et la poupée à l'intérieur est de plus en plus petite jusqu'à ce que vous arriviez au plus petit enfant, qui ne s'ouvre pas. Lorsque vous atteignez le plus petit enfant, vous inversez le processus en fermant chaque poupée une par une dans l'ordre inverse.

Par ailleurs, nous utilisons tous les jours la récursivité lorsque nous définissons des mots ! En effet, nous utilisons des mots pour en définir d'autres, eux-mêmes étant définis par d'autres mots !

La récursivité en programmation

En programmation, il s'agit d'une fonction qui fait référence à elle-même. Deux fonctions peuvent s'appeler l'une l'autre, on parle alors de récursivité croisée.

Essayons de retranscrire l'exemple des poupées russes à l'aide d'une fonction récursive. Nous allons alors compter le nombre de poupées à l'aide d'une fonction récursive.

Nous devons tout d'abord créer la fonction qui prend en paramètre une poupée, qu'elle contienne ou non une autre poupée :

```
Algorithme nombre_de_poupées(poupée)
Début


Fin
```

Nous pouvons maintenant construire le corps de la fonction. Si la poupée en paramètre contient une autre poupée, nous retournons 1 pour compter cette poupée,

et nous appelons de manière récursive la même fonction pour vérifier si l'autre poupée contient aussi une autre poupée.

```
Algorithme nombre_de_poupées(poupée)
Début
    Si poupée contient une autre poupée :
        Retourner 1 + nombre_de_poupées(poupée à l'intérieur)
    Fin Si
Fin
```

Pour terminer, nous devons trouver un moyen **d'arrêter** cette fonction récursive, sinon nous aurons **une boucle infinie**. Ainsi, si la poupée ne contient pas de poupée, nous allons retourner 1, simplement pour comptabiliser cette poupée.

Nous avons donc cette fonction finale :

```
Algorithme nombre_de_poupées(poupée)
Début
    Si poupée contient une autre poupée :
        Retourner 1 + nombre_de_poupées(poupée à l'intérieur)
    Sinon
        Retourner 1
    Fin Si
Fin
```

Comme pour les boucles, il faut toujours penser à une condition qui permette **d'arrêter** les appels récursifs.

La suite de Fibonacci

Avez-vous déjà passé un entretien d'embauche technique pour travailler en tant que développeur·se, par exemple ? Si oui, la suite de Fibonacci vous parlera certainement ! Il s'agit d'un des problèmes les plus couramment posés en entretien.

La suite de Fibonacci est une liste de nombres entiers. Elle commence généralement par les nombres 0 et 1 (parfois 1 et 1). On appelle un nombre de cette liste un terme. Chaque terme est la somme des deux termes qui le précèdent.

Par exemple, si la suite de Fibonacci débute ainsi : 0, 1, 1, 2, 3, 5, 8, 13, 21, etc., vous voyez que 0 + 1 donne 1, que 1 + 2 donne 3, que 3 + 5 donne 8, et ainsi de suite.

Il existe plusieurs manières de résoudre le problème lié à la suite de Fibonacci. Prenez quelques minutes pour réfléchir aux différents algorithmes que vous pourriez inventer.

Pour résoudre ce problème, nous allons utiliser une méthode récursive et une méthode itérative, afin que vous puissiez comparer les deux possibilités !

La méthode par algorithme récursif naïf

Un algorithme naïf désigne un algorithme "simple", très proche de notre pensée quotidienne. Concrètement, il s'agit de la première manière qui vous vient à l'esprit pour résoudre un problème.

Dans notre cas, nous pouvons dire :

```
Algorithme fibonacci(n)
Début
    Si n <= 1:
        Retourner 1
    Sinon
        Retourner fibonacci(n - 1) + fibonacci(n - 2)
    Fin Si
Fin
```

Si on appelle la fonction `fibonacci` avec le paramètre 3 `fibonacci(3)` , la fonction renverra **3**. Mais pourquoi ?

- `fibonacci(3)` : appelle les fonctions `fibonacci(2)` et `fibonacci(1)` :
  - `fibonacci(1)` : cette fonction s'arrête et renvoie 1.
  - `fibonacci(0)` : cette fonction s'arrête et renvoie 1.
  - Donc `fibonacci(2)` renvoie 1 + 1.
  - `fibonacci(2)` : appelle les fonctions `fibonacci(1)` et `fibonacci(0)` .
  - `fibonacci(1)` : cette fonction s'arrête et renvoie 1.
- Finalement on additionne ce que retournent `fibonacci(2)` et `fibonacci(1)` , soit 2 + 1.

Donc `fibonacci(3)` = 3.
Le problème dans ce cas est que nous consommons beaucoup de mémoire. En effet, le calcul est exponentiel : il demande deux calculs différents pour se calculer lui-même.

Quand nous calculons la valeur d'un nombre, nous réalisons les trois opérations suivantes :

- calcule fibonacci(n-1) [qui lui-même va calculer fibonacci(n-1) et ainsi de suite jusqu'à arriver au chiffre 1] et garde la valeur en mémoire ;
- calcule fibonacci(n-2) [fais-en de même à chaque fois jusqu'à arriver à 1] et garde la valeur en mémoire ;
- enfin, ajoute les deux précédentes valeurs.

Nous voyons par conséquent que ce calcul est exponentiel : chaque nouveau nombre demande deux fois plus de mémoire que son précédent. Sa complexité est de O(2^n).

Peut-on trouver une manière moins consommatrice de le calculer ?
Nous avons sans doute une solution toute trouvée avec un algorithme linéaire !

La méthode par algorithme linéaire

Et si vous calculiez deux valeurs consécutives à la suite ? Cela serait mieux, non ?

Nous pouvons ainsi écrire la fonction suivante :

```
Algorithme fibonacci(n)
Variable
    a ← 0 : ENTIER
    b ← 1 : ENTIER
    c ← 0 : ENTIER
Début
    Pour i allant de 1 jusqu'à n:
        c ← a + b
        a ← b
        b ← c
    Retourner b
Fin
```

À présent, chaque nouveau nombre n'a plus besoin de deux niveaux d'opération pour le générer, mais d'un seul. Il est donc linéaire et sa complexité est de O(n).

Nous voyons ici que, dans bien des cas, nous préférerons la version itérative, car elle est moins consommatrice en mémoire. La **récursivité** n'est pas la solution à tous les problèmes.

Voici les étapes importantes pour programmer une fonction récursive :

1. Décomposer le problème en un ou plusieurs sous-problèmes du même type. On résout les sous-problèmes par des appels récursifs.
2. Les sous-problèmes doivent être de taille plus petite que le problème initial.
3. Enfin, la décomposition doit en fin de compte conduire à un élémentaire qui, lui, n'est pas décomposé en sous-problèmes (c'est la condition d'arrêt).

Vous vous rappelez quand nous avons décomposé **l'appel** de chaque fonction récursivement pour le cas de la suite de Fibonacci ? Nous avons pu identifier l'ordre d'appel de chaque fonction pour identifier le résultat final de ces appels récursifs. En fait, l'ordinateur va **automatiquement empiler** chaque appel de fonctions afin de connaître **l'ordre** des appels, et avoir des **informations** au sujet des fonctions **actives**. C'est ce qu'on appelle la **pile d'exécution.**

Les piles d'exécution

L’ordinateur doit retenir le résultat de tous les calculs récursifs avant de finir sa boucle. Il utilise alors ce que nous appelons une "pile d’exécution" (en anglais, *stack*). Celle-ci est gérée automatiquement par le système.

Prenons par exemple le cas cette fonction `main` dans le labyrinthe :

```
Algorithme main()
Variable
    joueur_position_x ← 0 : ENTIER
    joueur_position_y ← 0 : ENTIER
    arrivée_position_x ← 5 : ENTIER
    arrivée_position_y ← 5 : ENTIER
    clés[3] : TABLEAU CHAÎNE DE CARACTÈRES
Début
    Tant Que joueur_position_x != arrivée_position_x ET joueur_position_y != arrivée_position_y ET
clés.taille() != 3:
        déplacement(joueur_position_x, joueur_position_y)
        ramasser(clés, joueur_position_x, joueur_position_y)
    Fin Tant Que
Fin
```

Lorsque la fonction main est appelée, on la place dans la pile d’exécution :

```
main()
```

et elle y est tant qu’elle est **active**, c’est-à-dire qu’elle n’est pas terminée.

Ensuite dans la fonction `main` , nous appelons une première fois la fonction `déplacement` dans la boucle `Tant que` , on l’ajoute ainsi dans la pile d’exécution, au-dessus de la fonction `main` .

```
déplacement(joueur_position_x, joueur_position_y)
main()
```

Tout de suite après, la fonction `déplacement` se termine, donc on la déplie :

```
main()
```

Mais dans la ligne suivante, on appelle la fonction `ramasser` donc on l’empile. On a donc maintenant :

```
ramasser(clés, joueur_position_x, joueur_position_y)

main()
```

Et la fonction `ramasser` , de la même manière, sera dépilée lorsqu’elle sera terminée, et ainsi de suite tant que la boucle est active. Lorsqu’elle se termine, la fonction main se termine aussi. Et donc on retire aussi la fonction `main` de la pile d’exécution. Quand la pile est vide, le programme est terminé !
La pile d’exécution peut aussi se noter "pile d’appels".
Dans le cas d’un appel récursif, c’est exactement ce qui se passe !

Considérons la fonction récursive suivante :

```
Algorithme compteur(n)
    Si n > 0 :
        compteur(n - 1)
    Fin Si
    Afficher n
Fin
```

empile

Lorsque nous appelons la fonction `compteur` avec `n=2` , nous avons la pile d'appels suivante :

empile compteur(2)	empile compteur(1)	empile compteur(0)
compteur(2)	compteur(1)	compteur(0)
	compteur(2)	compteur(1)
		compteur(2)

Tant qu'il y a des appels récursifs de la fonction `compteur` , on les empile dans la pile d'exécution.

dépile

Lorsque la dernière fonction récursive est appelée, l'ordinateur "dépile" l'ensemble des fonctions présentes dans la pile d'exécution. Autrement dit, il va chercher dans la pile le dernier élément enregistré, et ainsi de suite jusqu'à arriver en bas de la pile. Bien pratique !

dépile compteur(0)	dépile compteur(1)	dépile compteur(2)
compteur(0)	compteur(1)	
compteur(1)	compteur(2)	
compteur(2)		
Affiche 0	Affiche 1	Affiche 2

À vous de jouer !

Contexte

Pour finaliser notre labyrinthe, nous souhaitons ajouter une dernière fonctionnalité. Nous allons créer des cases pièges, qui bloqueront le joueur pendant 10 secondes à chaque fois que le joueur passera dessus.

Consigne

Vous allez pour cela créer une fonction récursive qui bloquera le joueur pendant n secondes, et qui affichera un compte à rebours du temps restant pour être débloqué. Vous appellerez cette fonction `freeze` .

Considérez que la fonction `attendre` permet de mettre en pause le jeu pendant un certain nombre de secondes. Le nombre de secondes est passé en paramètre de la fonction.

Votre objectif :

- Définir une fonction `freeze` qui prend en paramètre le nombre n de secondes à bloquer.
- Afficher le nombre de secondes restantes.
- Mettre en pause le jeu 1 seconde.
- Appeler récursivement la fonction `freeze` .
- Spécifier la condition d'arrêt de la fonction récursive.

Vérifiez votre travail

Voici le résultat à obtenir à l'issue de l'exercice :

```
Algorithme freeze(n)
    Si n == 0 :
        arrêter l'appel recursive
    Fin Si

    Afficher n
    attendre(1)
    freeze(n - 1)
Fin
```

En résumé

- Une fonction récursive est une fonction qui s'appelle elle-même pendant son exécution.
- Les étapes pour construire une fonction récursive :
    - Décomposer le problème en un ou plusieurs sous-problèmes du même type. On résout les sous-problèmes par des appels récursifs.
    - Les sous-problèmes doivent être de taille plus petite que le problème initial.
    - Enfin, la décomposition doit en fin de compte conduire à un élémentaire qui, lui, n'est pas décomposé en sous-problèmes (c'est la condition d'arrêt).
- La pile d'exécution sert à enregistrer des informations au sujet des fonctions actives dans un programme informatique.

C'est bon, c'est promis, cette fois c'est vraiment terminé ! Bravo pour l'ensemble de votre travail tout au long du cours. Et je vous dis à tout de suite dans le dernier dernier quiz de ce cours pour valider vos connaissances. Je sais que vous adorez les quiz !