Introduction à l'informatique
Algorithmes sur des tableaux

Benjamin Monmege

2021/2022

1 Définition d'un tableau

Un tableau est la manière la plus simple de ranger au même endroit un ensemble de données. Il est composé d'un ensemble de cases accolées les unes aux autres contenant les éléments dans un certain ordre. Par exemple, voici un tableau contenant 5 cases (on dit que le tableau est de longueur 5) :

17	64	5	1	38

Voici un tableau de caractères de longueur 10 :

'a'	'l'	'g'	'o'	'r'	'i'	't'	'h'	'm'	'e'

Plus généralement, voici un tableau t de longueur $n \in \mathbf{N}$ quelconque :

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$\cdots$	$t[n-2]$	$t[n-1]$

qu'on notera parfois de la façon suivante : $\big[t[0], t[1], t[2], t[3], \ldots, t[n-2], t[n-1]\big]$. Classiquement, on commence à numéroter les cases d'un tableau à partir de 0, de sorte que la dernière case a le numéro $n-1$ (et pas n) : la i-ième case du tableau t (avec $0 \leq i \leq n-1$) contient une valeur qu'on note $t[i]$. On peut donc voir un tableau de longueur n contenant des éléments d'un ensemble E (des entiers, des caractères, etc.) comme une application $t \colon \{0, 1, \ldots, n-2, n-1\} \to E$ associant à chaque indice i une valeur notée $t[i]$ dans E.

Dans les algorithmes que nous allons écrire, on pourra donc :

— récupérer la longueur d'un tableau `t` à l'aide de
```
n := longueur(t)
```
— initialiser un tableau vide de longueur `n` à l'aide de
```
t := tableau vide de longueur n
```
— accéder au contenu de la `i`-ème case d'un tableau `t` à l'aide de `t[i]`

— modifier le contenu de la `i`-ème case d'un tableau `t` à l'aide de
```
t[i] := x
```

2 Rechercher dans un tableau

Un tableau peut aussi permettre de stocker simplement un ensemble de données telles que :

— un paquet de cartes à jouer ;
— des mots (avec leur définition) dans un dictionnaire ;
— ou des noms (avec leur adresse et numéro de téléphone) dans un annuaire.

Une opération cruciale lorsqu'on stocke un tel ensemble de données est de pouvoir tester si un élément appartient à l'ensemble ou non.

2.1 Recherche séquentielle

Imaginons pour commencer qu'on souhaite vérifier si notre paquet de cartes à jouer contient un joker ou pas. Pour cela, on a peu d'autres choix que de rechercher le joker en passant les cartes du paquet en vue l'une après l'autre, en commençant par un bout du paquet.

Dans le contexte des tableaux, cela revient à rechercher un élément dans le tableau en parcourant le tableau (de gauche à droite par exemple) et en s'arrêtant dès lors qu'on a trouvé l'élément : cet algorithme s'appelle la *recherche séquentielle* (ou recherche par balayage), puisqu'on visite les éléments du tableau séquentiellement, c'est-à-dire les uns à la suite des autres. On peut écrire cet algorithme de la façon suivante :

```
fonction rechercher_séquentiel(tableau, élément):
    n := longueur(tableau)
    Pour i de 0 à n−1 faire
        Si tableau[i] = élément alors
            retourner(i)
        FinSi
    FinPour
    retourner(−1)       # élément pas trouvé !
```

L'algorithme renvoie l'indice d'une case du tableau contenant l'élément recherché, si un tel indice existe, et -1 sinon : dès que l'instruction `retourner(i)` est exécutée, la fonction s'arrête et ne poursuit donc pas son exploration. Au contraire, si la boucle `Pour` s'est exécutée entièrement sans jamais trouver l'élément recherché, alors on peut renvoyer -1 puisqu'on est alors sûr que l'élément recherché ne se trouve pas dans le tableau.

Imaginons désormais qu'on recherche un mot dans un dictionnaire, ou un nom dans un annuaire. A priori, vous n'utilisez pas l'algorithme de recherche séquentielle dans ce cas. Essayons de comprendre pourquoi en estimant la *complexité* de cet algorithme. Comment faire cela ? Une première possibilité consisterait à déclencher un chronomètre en même temps que le début de l'exécution de l'algorithme (par un humain ou un ordinateur), afin de voir le temps qu'il met avant de renvoyer le résultat. Cette première solution est malheureusement peu précise car pas nécessairement reproductible : l'humain ou l'ordinateur qui exécute l'algorithme ne mettra pas toujours le même temps, selon qu'il est plus ou moins en forme, qu'il a plus ou moins d'autres choses à penser ou à faire en même temps. De plus, on arriverait alors pas à comparer les complexités d'algorithmes dont l'un serait exécuté par un humain et l'autre par un ordinateur, ou par deux ordinateurs différents.

Pour remédier à ce problème, on utilise une autre méthode pour estimer la complexité d'un algorithme : on compte plutôt le nombre d'*opérations élémentaires* que l'algorithme exécute *dans le pire des cas* pour des entrées de taille fixée.

— Une opération élémentaire, cela correspond *grosso modo* à une ligne de pseudo-code : de manière générale, il est important de se fixer un ensemble d'opérations élémentaires qu'on comptabilise ensuite une par une lors de l'exécution de l'algorithme.
— La définition précise *dans le pire des cas* puisqu'on voudrait pouvoir décrire la complexité de l'algorithme sur toutes les entrées possibles d'une taille fixée : mais il se peut

que, pour certaines entrées, le résultat soit très rapide à calculer, et que, pour d'autres, ce soit beaucoup plus long. Pour régler ce problème, on considère donc le pire des cas possibles, qui borne donc la complexité de toutes les instances d'une taille fixée.

Dans l'algorithme de recherche séquentielle, l'affectation de la longueur du tableau dans la variable `n` peut être vue comme une opération élémentaire, de même que le test d'égalité entre le contenu de la case d'indice `i` et l'élément à chercher, ou le fait de retourner un résultat. On cherche donc à estimer le nombre de telles opérations élémentaires lorsque l'algorithme s'exécute sur un tableau de longueur n arbitraire (n étant supposé grand).

- Dans tous les cas, on affecte à la variable `n` la longueur du tableau, ce qui coûte une opération élémentaire.
- Une itération de la boucle `Pour` effectue une opération élémentaire (test d'égalité), et au plus une fois au total le retour de la valeur de sortie.
- Puisque dans le pire des cas (c'est-à-dire lorsqu'on cherche un élément qui n'apparait pas dans le tableau), on exécute cette boucle n fois (une fois par case du tableau), au total, on exécute donc n opérations (sans compter le retour).
- Finalement, dans tous les cas, on exécute soit le retour au sein de l'itération, ou le retour en dehors de l'itération, qui coûte donc toujours une opération élémentaire.

Au total, on exécute donc $1+n+1$ opérations élémentaires dans le pire des cas, c'est-à-dire $n+2$ opérations élémentaires.

Considérons désormais le point de vue d'un tableau « très long », c'est-à-dire avec n très grand. Dans ce cas, il n'est pas très intéressant de distinguer $n+2$ de n : on préfère donc conserver uniquement l'*ordre de grandeur* de la complexité. À ce titre, introduisons la notation de Landau (du nom d'Edmund Landau qui l'a introduite) permettant de comparer deux telles *suites*.

Définition 1. Soient $u = (u_n)_{n\in\mathbf{N}}$ et $v = (v_n)_{n\in\mathbf{N}}$ deux suites à valeurs entières : les entiers u_n et v_n décrivent donc des nombres d'opérations élémentaires pour l'exécution d'un algorithme dans le pire des cas sur une entrée de taille n. On dit que u est en $O(v)$ (qui se lit « grand o de v », comme la lettre de l'alphabet...) s'il existe un entier N et une constante $c > 0$ tels que pour tout $n \geq N$, on a $u_n \leq cv_n$. Dans la suite, on s'autorise à écrire que u_n est en $O(v_n)$.

Par exemple, on obtient alors bien que $n+2$ est en $O(n)$: en effet, pour $N = 2$ et $c = 2$, on a bien que pour tout entier $n \geq 2$, $n + 2 \leq n + n = 2n = cn$.

Dans ce cours, on considèrera donc que « $n+2$ ou n, c'est pareil ! ». On peut aller plus loin et montrer qu'en fait, « $2n$ ou n, c'est pareil ! » : en effet, pour $N = 0$ et $c = 2$, on obtient trivialement que pour tout $n \geq 0$, $2n \leq cn$. En terme d'informatique, cela veut dire qu'un algorithme A et un autre algorithme B qui va deux fois plus vite que A ne sont généralement pas distingué en terme de complexité dans le pire des cas : fondamentalement, c'est parce qu'il suffit d'avoir une machine deux fois plus puissante pour que B devienne aussi performant que A.

Au contraire, n^2 n'est pas un $O(n)$, c'est-à-dire que n^2 grossit beaucoup plus vite que n. Vous pouvez le vérifier en montrant que vous ne pouvez pas trouver N et c vérifiant la définition lorsque $u_n = n^2$ et $v_n = n$. Par contre, on a bien que n est un $O(n^2)$, mais ce n'est pas très intéressant : cela veut juste dire que n grossit moins vite que n^2, mais on y perd donc beaucoup en faisant cette approximation... L'ordre de grandeur d'une complexité de la forme $a_k n^k + a_{k-1}n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$ (avec k fixé et $a_k \neq 0$) est donc en $O(n^k)$, à savoir le plus grand terme dans l'écriture polynomiale de la complexité.

Revenons alors à l'algorithme de recherche séquentielle dont on a vu qu'il exécutait dans le pire des cas $n + 2$ opérations élémentaires. On dira donc qu'il a une complexité en $O(n)$, *linéaire* en la longueur du tableau en entrée. Si l'on recherche un mot dans un dictionnaire contenant 32 000 mots par exemple, cela demande donc un nombre d'opérations de l'ordre de 32 000 : si on le fait « à la main », même si on pouvait exécuter 10 opérations à la seconde, il nous faudrait alors 53 minutes pour rechercher un mot dans le dictionnaire...

On souhaite donc faire mieux en utilisant l'information qu'un dictionnaire, ou un annuaire, n'est pas un tableau quelconque. En effet, les mots du dictionnaire, ou les noms de l'annuaire, sont triés par ordre alphabétique croissant. On dit donc d'un tableau qu'il est trié si ses éléments sont classés par ordre croissant : le tableau $[1, 4, 5, 8]$ est donc trié, contrairement au tableau $[8, 3, 4, 5]$.

2.2 Recherche dichotomique dans un tableau trié

Profitons donc au mieux du fait qu'on recherche dans un tableau trié (dictionnaire ou annuaire) pour décrire un algorithme de recherche a priori plus performant. Lorsqu'on cherche dans un dictionnaire, on ne regarde pas les mots les uns à la suite des autres en commençant par la lettre A, d'autant plus si on recherche la définition du mot « pingouin »... On va plutôt essayer d'estimer la position du mot dans le dictionnaire, ouvrir le dictionnaire à la page correspondante, puis prendre une décision pour continuer notre recherche à gauche de la page courante, ou à droite. On continue ensuite ainsi de suite jusqu'à avoir trouvé le mot, ou être sûr que le mot n'existe pas dans ce dictionnaire.

Ce type de recherche dans un tableau trié s'appelle la *recherche dichotomique*, du grec $\delta\iota\chi\sigma\tau\sigma\mu\acute{\iota}\alpha$ qui signifie « division en deux parties égales ». Dans le cas général, on part donc d'un tableau trié et on commence par comparer l'élément qu'on recherche avec l'élément qui se trouve au milieu (environ) du tableau. Partons du tableau trié d'entiers

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

et cherchons-y l'élément 26. On commence par regarder la case du milieu : le tableau étant de longueur 16, le milieu du tableau correspond à la case d'indice 7 (les cases sont indexées de 0 à 15), qui héberge l'élément 14. Il est différent de 26 : il faut donc continuer notre recherche. Par ailleurs, 14 est strictement inférieur à 26 : puisque le tableau est trié, cela implique que l'élément 26 ne peut pas se trouver dans la portion du tableau à gauche de l'élément 14. On peut donc se restreindre à rechercher 26 dans la partie non grisée du tableau :

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Parmi les 8 éléments restants, on poursuit en comparant 26 avec l'élément du milieu du tableau restant, la case contenant 27. Ces deux éléments sont toujours différents, mais cette fois, la comparaison nous apprend que l'élément 26 ne peut pas se trouver dans la portion du tableau à droite de l'élément 27. On se restreint ainsi à la portion non grisée du tableau

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Il reste une portion de longueur 3, dont 25 est l'élément du milieu. Une fois de plus, on supprime la partie de gauche de la portion restante pour ne laisser plus qu'une seule case qui abrite l'élément 26 que nous recherchions :

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Sur le même tableau en entrée, si l'on recherche l'élément 6, voici les étapes par lesquelles on passe :

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Cela nous permet de répondre avec certitude que le tableau de contient pas l'élément 6.

Voici une écriture sous forme de pseudo-code de l'algorithme que nous venons d'exécuter :

```
fonction rechercher_dichotomique(tableau, élément) :
    n := longueur(tableau)
    début := 0
    fin := n−1
    Tant que début ≤ fin faire
        milieu := ⌊(début + fin)/2⌋
        Si tableau[milieu] = élément alors
            retourner(milieu)
        Sinon Si tableau[milieu] < élément alors
            # l'élément est à droite
            début := milieu+1
        Sinon  # l'élément est à gauche
            fin := milieu−1
        FinSi
    FinTantQue
    retourner(−1)
```

On y maintient deux variables `début` et `fin` qui conservent en mémoire l'indice de la première et de la dernière case de la portion de tableau qu'il reste au cours de la recherche. On initialise donc ces deux variables avec les première et dernière cases du tableau respectivement, d'indice 0 et $\texttt{n} - 1$. La recherche se termine lorsqu'on est convaincu que l'élément recherché ne se trouve pas dans le tableau, c'est-à-dire lorsqu'il ne reste plus aucune portion de tableau à parcourir : c'est le cas lorsque l'indice `début` est strictement supérieur à `fin`. Il faut donc continuer la rechercher *tant que* cette condition n'est pas vérifiée, c'est-à-dire *tant que* `début` est inférieur ou égal à `fin`. Dans ce cas, il faut alors calculer l'indice de la case du milieu : le milieu de l'intervalle $[a, b]$ est le nombre $(a+b)/2$. Puisqu'on veut obtenir un indice de case qui est un entier, on choisit le quotient dans la division euclidienne de $a + b$ par 2 ; autrement dit, la partie entière de $(a + b)/2$, qu'on note $\lfloor (a + b)/2 \rfloor$. On stocke l'indice de la case du milieu dans une variable `milieu`. On peut alors comparer le contenu de cette case avec l'élément à rechercher : si on le trouve, on s'arrête en retournant l'indice `milieu` ; sinon, on effectue une comparaison supplémentaire pour savoir si l'élément a des chances d'apparaître à gauche ou à droite du milieu. Selon le cas, on met à jour soit le début de la portion, soit la fin de la portion.

Un algorithme est la description *non ambiguë* d'une séquence *finie* d'instructions permettant de résoudre un problème (informatique) ou d'obtenir un résultat. Est-ce que la recherche dichotomique est bien un *algorithme* en ce sens ?

2.2.1 Terminaison

La définition insiste sur la finitude : un algorithme doit terminer après un nombre fini d'étapes. Est-on bien sûr que c'est le cas ici ? Dans cet algorithme, c'est l'utilisation d'une boucle `Tant que` qui pourrait faire échouer cette condition de terminaison de l'algorithme : il se pourrait en effet que la condition d'arrêt de la boucle ne soit jamais vérifiée, l'algorithme bouclant alors à l'infini. Pourquoi est-on certain ici que cela n'arrivera pas ? Il s'agit de s'assurer que la boucle `Tant que` termine, c'est-à-dire que le test `début` $\leq$ `fin` finit par devenir faux. Autrement dit, il faut s'assurer qu'à un moment de l'algorithme, on finit par avoir `début` $>$ `fin`. C'est le cas, puisqu'à chaque étape de la boucle `Tant que` :

— soit `début` augmente strictement ;
— soit `fin` diminue strictement.

Ainsi, à chaque itération `fin` $-$ `début` diminue strictement : puisque c'est un entier, il finit par devenir négatif. Le test `début` $\leq$ `fin` finit donc par être faux.

2.2.2 Correction

L'utilisation d'une boucle `Tant que` complique également notre intuition sur la *correction* de l'algorithme : est-on sûr qu'il trouve bien un élément dès lors que celui-ci apparaît dans le tableau et qu'il renvoie -1 uniquement lorsque l'élément ne s'y trouve pas ? S'assurer que l'algorithme est correct, c'est montrer qu'il fait bien ce qu'il est sensé faire :

— si l'élément qu'on cherche se trouve dans le tableau trié, alors l'algorithme doit renvoyer l'indice d'une case du tableau où se trouve l'élément ;
— si l'élément qu'on cherche ne se trouve pas dans le tableau trié, alors l'algorithme doit renvoyer -1 : ceci est facile à montrer puisque l'algorithme ne peut renvoyer autre chose que -1 que s'il a trouvé un indice du tableau hébergeant l'élément recherché.

Concentrons-nous donc sur la première propriété. Pour s'assurer de cette propriété, on écrit un *invariant de boucle*, c'est-à-dire une propriété qui est vraie initialement, et qui reste vraie tout au long de l'algorithme. Ici, l'invariant de boucle est le suivant : si on suppose que le tableau en entrée est trié par ordre croissant et contient l'élément recherché, alors à tout moment de l'exécution de l'algorithme on est sûr qu'il existe un indice i entre `début` et `fin` tel que `tableau`$[i]$ $=$ `élément`.

C'est vrai en début de boucle puisque `début` $= 0$ et `fin` $=$ `n` $- 1$ donc la portion est le tableau tout entier. On peut ensuite se convaincre que l'exécution d'une itération de la boucle `Tant que` préserve la propriété, puisqu'on a simplement retiré une portion du tableau où l'on est sûr que l'élément ne se trouve pas (c'est ici qu'on utilise le fait que le tableau est trié !).

Par le principe de récurrence (sur le nombre de tours de boucle), l'invariant de boucle est donc vrai pendant toute l'exécution : en particulier, il est vrai lorsque l'on sort de la boucle `Tant que` (ce qui est sûr d'arriver puisque l'algorithme termine). Mais si l'on sort de la boucle sans avoir jamais retourné d'indice, on a alors `début` $>$ `fin` et il n'existe plus aucun indice i entre `début` et `fin`, contredisant la propriété qu'on a démontré.

2.2.3 Complexité

Pour terminer l'étude de cet algorithme, il ne nous reste plus qu'à étudier la complexité, pour la comparer avec celle de la recherche séquentielle qu'on a étudié avant.

Supposons que le tableau a une longueur $n = 2^p$ pour simplifier le calcul. On cherche donc le nombre d'opérations élémentaires exécutées par l'algorithme dans le pire des cas sur un

tableau de longueur 2^p. Le pire des cas intervient lorsqu'on ne trouve pas l'élément dans le tableau puisque la recherche ne s'interrompt alors pas prématurément. Ensuite, remarquons que chaque itération de la boucle `Tant que` exécute 5 opérations élémentaires dans le pire des cas :

1. le test `début` $\leq$ `fin` ;
2. le calcul du nouveau milieu pour l'affectation de la variable `milieu` ;
3. le test `tableau`[`milieu`] = `élément` : le pire des cas se produit si ce test n'est pas satisfait ;
4. le test `tableau`[`milieu`] $<$ `élément` ;
5. l'affectation de la nouvelle valeur de `début` ou `fin` selon le cas.

Il reste donc à connaître le nombre d'itérations de la boucle `Tant que`. On peut la déduire en fonction de la longueur de la portion restante du tableau à explorer : cette longueur vaut toujours `fin` $-$ `début` $+ 1$. À chaque itération, la taille de la portion restante de tableau est au moins divisée par deux (d'où le nom de *dichotomie* !). Par conséquent,

— après 0 itération (au début), la longueur de la portion restante est 2^p ;
— après 1 itération, la longueur de la portion restante est au plus 2^{p-1} ;
— après 2 itérations, la longueur de la portion restante est au plus 2^{p-2} ;
— …
— après k itérations, la longueur de la portion restante est au plus 2^{p-k} ;
— …
— après $p+1$ itérations, la longueur de la portion restante est au plus $2^{p-(p+1)} = 1/2$, et puisqu'elle est entière, la longueur est nulle ; autrement dit le tableau est vide.

On est donc assuré qu'il y a au plus $p+1$ itérations de la boucle `Tant que`. Le nombre d'opérations élémentaires exécutées par la boucle dans le pire des cas est donc d'au plus $5 \times (p+1)$.

Par ailleurs, en dehors de la boucle, l'algorithme exécute au plus 4 opérations élémentaires :

1. l'affectation de `n` ;
2. l'affectation de `début` ;
3. l'affectation de `fin` ;
4. le retour de la valeur de sortie.

Le nombre total d'opérations élémentaires est donc au plus $4+5\times(p+1)$. Puisque $n = 2^p$, on a $p = \log_2 n$, donc la complexité est en $4 + 5 \times (\log_2 n + 1)$. Pour n supérieur à 2 (de sorte que $1 \leq \log_2 n$), on a $4 + 5 \times (\log_2 n + 1) \leq 14 \times \log_2 n$. Ainsi, la complexité de la recherche dichotomique est de l'ordre de $O(\log_2 n)$, logarithmique en la longueur n du tableau.

Pour s'apercevoir de la différence cruciale entre une complexité linéaire (comme la recherche séquentielle) et une complexité logarithmique (comme la recherche dichotomique), observons que $\log_2(32\ 000) \approx 15$ ce qui veut dire que la méthode de la recherche dichotomique permet de trouver n'importe quel mot du dictionnaire contenant 32 000 mots en regardant au plus 15 pages du dictionnaire : à raison d'une page par seconde, cela donne le mot en 15 secondes au plus (à comparer aux 53 minutes qu'on avait calculé précédemment pour la recherche séquentielle). Lorsque n devient encore plus grand, l'écart est de plus en plus important, puisque la suite $(\log_2 n)_{n\in\mathbf{N}^*}$ croit exponentiellement moins vite que la suite $(n)_{n\in\mathbf{N}^*}$. Même pour un tableau (trié) comprenant autant de cases que le nombre d'atomes dans l'univers visible (environ 10^{79}), on parvient tout de même à rechercher un élément en un nombre d'opérations élémentaires de l'ordre de $\log_2(10^{79}) \approx 263$.

3 Tri par insertion

On a vu que pour rechercher un élément dans un tableau, la complexité était bien meilleure dès lors que le tableau était trié (c'est-à-dire que ses éléments sont rangés dans un ordre croissant). Il est donc naturel de se demander comment faire en sorte de trier un tableau et de la complexité nécessaire pour cela. Ce problème très naturel de tri d'éléments apparaît à de nombreuses reprises en informatique : par exemple, lorsqu'on a voulu visualiser la liste des restaurants à proximité par note moyenne décroissante, c'est un tri (par ordre décroissant, plutôt que croissant...) que l'on exécute.

Commençons par considérer la situation où l'on cherche à ranger dans sa main des cartes à jouer suivant un ordre précis (afin de séparer les couleurs, puis de ranger les cartes par ordre croissant dans chaque couleur, par exemple). Pour simplifier, considérons un cas particulier où nous n'avons que des cartes de carreau. Prenez une dizaine de cartes et rangez-les par ordre croissant : analysez alors votre façon de faire...

Je décris ici ma façon de faire lorsqu'il s'agit de trier un nombre conséquent de cartes. Je laisse le tas de cartes à trier face contre la table et prend les cartes les unes après les autres. Je prends la première carte dans ma main. Je retourne ensuite la seconde carte que je viens placer au bon endroit (avant ou après la première carte) dans ma main. Je retourne alors la troisième carte que je dois de même insérer au bon endroit dans ma main. Lorsque j'arrive à la treizième carte, il devient plus difficile de savoir où je dois l'insérer : si je décompose à nouveau ma routine, je vois que je *scanne* la suite des cartes de droite à gauche jusqu'à trouver l'endroit où je dois insérer la nouvelle carte. Essayez d'imiter ma méthode afin de classer un paquet de treize cartes de carreau, en décomposant bien chaque étape en opérations les plus simples possibles.

Cette procédure de tri s'appelle le *tri par insertion* du fait qu'on ne fait qu'insérer les éléments les uns après les autres au bon endroit dans la portion triée. On peut également l'appliquer pour trier un tableau. On suppose donc qu'on a en entrée de l'algorithme un tableau non trié, par exemple, le tableau $[10, 8, 2, 5, 13]$. On considère les éléments de gauche à droite, en cherchant à chaque fois à les placer au bon endroit dans la portion triée qui sera la partie gauche du tableau. Initialement, on considère donc le premier élément du tableau (10) qui est bien placé puisque c'est le seul élément considéré jusqu'alors. La portion triée est donc $[10, ...$ et il reste le tableau $...8, 2, 5, 13]$ à considérer. On regarde ensuite le second élément (8) puis on le compare à l'élément à sa gauche, afin de l'insérer au bon endroit dans la portion triée : ici $8 < 10$ donc il faut échanger les deux éléments, ce qui termine l'insertion de 8 à sa place. On se retrouve alors avec la portion triée $[8, 10, ...$ et le reste du tableau $...2, 5, 13]$. Au coup suivant, on doit considérer l'élément 2 : il est inférieur à 10, donc on doit échanger sa place avec 10 (temporairement on obtient donc le tableau $[8, 2, 10, 5, 13]$), puis on le compare avec l'élément à sa gauche (8) pour arriver à la situation où la portion triée est $[2, 8, 10, ...$ et le reste $...5, 13]$. L'insertion de 5 se fait alors en deux étapes : on échange les éléments 10 et 5, puis les éléments 8 et 5, avant de voir que $2 < 5$ ce qui achève l'insertion de l'élément 5. Finalement, on considère l'élément 13 qui est directement supérieur à l'élément à sa gauche (10) stoppant dès le début son insertion. On termine donc avec le tableau trié $[2, 5, 8, 10, 13]$.

Voici une écriture sous forme de pseudo-code de cet algorithme :

```
fonction trier_par_insertion(tableau) :
    n := longueur(tableau)
    Pour i de 1 à n−1 faire
        x := tableau[i]
```

```
    # insérer x parmi les i premiers éléments
    j := i
    Tant que ((j > 0) et (x < tableau[j−1])) faire
        # décaler d'un élément
        tableau[j] := tableau[j−1]
        j := j−1
    FinTantQue
    # ici, x ≥ tableau[j−1] ou bien j= 0
    tableau[j] := x
FinPour
# le tableau est trié !
```

Exécutez l'algorithme sur le tableau $[8, 2, 10, 5, 13]$ pour bien comprendre comment il fonctionne : vous verrez qu'il ne fait pas exactement ce qui est dit au-dessus en s'épargnant des échanges de cases inutiles...

Notons également que cet algorithme ne retourne aucun résultat : on a effectivement choisi de faire un tri *en place*, c'est-à-dire qu'on ne renvoie pas un nouveau tableau, mais qu'on a modifié le tableau donné en entrée directement... Il est donc inutile de le retourner, puisqu'on a directement modifié le tableau dans la mémoire. Notons qu'en Python, le passage d'un tableau en argument d'une fonction se fait bien « par référence » et donc la modification au sein de la fonction se répercute bien sur la mémoire globale : attention, ce n'est pas le cas dans tous les langages de programmation !

Pour mieux comprendre cet algorithme (puis le comparer avec d'autres algorithmes, dans la suite), estimons sa complexité dans le pire des cas. Comme pour les algorithmes de recherche étudiés auparavant, il faut donc comptabiliser les opérations élémentaires.

Vu l'imbrication de boucles de cet algorithme, il convient de commencer par considérer les boucles les plus internes, c'est-à-dire le code qui est le plus « décalé à droite ». Considérons donc d'abord la boucle `Tant que` qui exécute deux opérations élémentaires à chaque itération (une affectation dans `tableau[j]` et une décrémentation de `j`), auxquelles il faut ajouter les opérations élémentaires nécessaires pour tester si l'on doit continuer la boucle ou pas : ce test (`(j> 0) et (x<tableau[j−1])`) requiert deux comparaisons qu'on comptabilise donc comme deux opérations élémentaires. Une itération réclame donc 4 opérations élémentaires. Il faut aussi considérer le dernier test qui fait sortir de la boucle `Tant que`, qui nécessite aussi 2 tests dans le pire des cas. Pour totaliser la complexité de cette boucle interne, il nous suffit donc de savoir le nombre de fois que cette boucle s'exécute. Dans le pire des cas, l'élément `x` est inférieur à tous les éléments de la portion triée, ce qui pousse alors à le décaler jusqu'au tout début du tableau : la variable `j` prend alors toutes les valeurs de `i` à 1, avant d'être égale à 0, auquel cas on sort de la boucle. Dans le pire des cas, on exécute donc `i` fois la boucle, générant donc au total $4\texttt{i} + 2$ opérations élémentaires.

On peut ensuite passer à la boucle `Pour` :

— on y affecte la variable `x`

— ainsi que la variable `j` ;

— on exécute la boucle `Tant que`, coûtant donc $4\texttt{i} + 2$ opérations élémentaires dans le pire des cas ;

— finalement, on modifie la valeur de `tableau[j]`.

Lors de l'itération correspondant à une valeur particulière de `i`, on exécute donc $1+1+4\texttt{i}+2+1 = 4\texttt{i} + 5$ opérations élémentaires. Ce nombre dépend de l'itération `i` : on ne peut donc pas simplement multiplier le nombre d'opérations par le nombre d'itérations. À la place, on fait la somme de toutes les contributions des itérations : le nombre total d'opérations élémentaires

exécutées au sein de la boucle `Pour` vaut donc

$$(4 \times 1 + 5) + (4 \times 2 + 5) + \cdots + (4 \times (n-1) + 5)$$

qu'on peut écrire sous la forme d'une somme qu'on décompose en deux sommes indépendantes :

$$\sum_{i=1}^{n-1}(4i+5) = \sum_{i=1}^{n-1}(4i) + \sum_{i=1}^{n-1} 5 = 4\sum_{i=1}^{n-1} i + 5 \times (n-1)$$

Il ne reste plus qu'à calculer la somme de gauche, qui n'est rien d'autre que la somme des $n-1$ premiers entiers dont on sait par ailleurs qu'elle vaut $(n-1)n/2$. On obtient donc un nombre d'opérations élémentaires égal à $2(n-1)n + 5(n-1) = 2n^2 + 3n - 5$.

Finalement, il faut aussi comptabiliser les opérations élémentaires en dehors de la boucle `Pour` : il n'y en a qu'une, l'affectation de la variable `n`. In fine, on exécute donc, dans le pire des cas, $2n^2 + 3n - 4$ opérations élémentaires. L'ordre de grandeur est donc en $O(n^2)$ (pour s'en convaincre ici, il suffit de remarquer que pour tout n supérieur à 3, $2n^2 + 3n - 4 \leq 2n^2 + 3n \times n = 5n^2$). L'algorithme de tri par insertion est donc un algorithme de complexité *quadratique* en la longueur du tableau à trier.

4 Tri par fusion

Proposons maintenant une autre façon de trier un tableau, utilisant une méthode tout à fait différente, consistant à diviser-pour-régner. Illustrons-la sur l'exemple du tableau

$$[3, 16, 14, 1, 12, 7, 10, 4, 5, 11, 15]$$

On va diviser le problème en deux sous-problèmes de la même forme : ici, il suffit de couper le tableau en deux, pour obtenir deux sous-tableaux $[3, 16, 14, 1, 12, 7]$ et $[10, 4, 5, 11, 15]$ de taille (presque) identique. Imaginons qu'on sache trier ces deux sous-tableaux : on obtient alors les tableaux $[1, 3, 7, 12, 14, 16]$ et $[4, 5, 10, 11, 15]$. Pour obtenir le grand tableau trié, il suffit donc de fusionner ces deux tableaux triés. Mais ceci est très facile à faire (imaginez que vous avez deux jeux de cartes triés et que vous voulez les fusionner en conservant le tri...). Il suffit de remarquer que le plus petit élément du grand tableau trié doit forcément être 1 ou 4 (les premiers éléments des deux petits tableaux triés), c'est donc 1. Pour obtenir la suite du grand tableau trié, on continue ce même raisonnement avec les deux petits tableaux $[3, 7, 12, 14, 16]$ et $[4, 5, 10, 11, 15]$: c'est 3 le plus petit élément des deux qui doit donc être à la suite de 1 dans le tableau. Puis 4 vient ensuite : on se retrouve alors avec deux petits tableaux de la forme $[7, 12, 14, 16]$ et $[5, 10, 11, 15]$. On continue ainsi jusqu'à avoir complètement fusionné les deux petits tableaux triés et obtenu le tableau

$$[1, 3, 4, 5, 7, 10, 11, 12, 14, 15, 16]$$

La question reste cependant entière : comment fait-on pour trier les deux petits tableaux $[3, 16, 14, 1, 12, 7]$ et $[10, 4, 5, 11, 15]$? La réponse est simple : « on recommence ! ». En effet, la tactique décrite ci-dessus pour trier le grand tableau peut être aussi utilisée pour traiter ces deux petits tableaux, de manière indépendante. On peut ainsi visualiser dans la FIGURE 1 le cheminement complet pour trier le grand tableau : la division en deux sous-tableaux est marquée par l'éclair rouge et les deux flèches bleues, puis la fusion des deux tableaux triés est représentée par les flèches orange.

FIGURE 1 – Représentation graphique du tri par fusion

Comment écrire dans du pseudo-code la formule magique « on recommence ! » ?!? La manière la plus simple de faire est la suivante :

```
fonction tri_fusion(tableau) :
    n := longueur(tableau)
    Si n ≤ 1 alors
        retourner(tableau)
    Sinon
        gauche := tableau[0..⌊n/2⌋]
        droite := tableau[⌊n/2⌋+1..n−1]
        gauche_trié := tri_fusion(gauche)
        droite_trié := tri_fusion(droite)
        retourner( fusionner(gauche_trié, droite_trié) )
    FinSi
```

On a utilisé la notation `tableau[0..⌊n/2⌋]` pour représenter la portion gauche du tableau constituée des cases d'indice $0, 1, \ldots, \lfloor n/2 \rfloor$ du tableau. Contrairement au tri par insertion, cette fonction retourne un tableau : elle n'est pas *en place*. On a également utilisé une fonction `fusionner` dont on ne fournit pas le pseudo-code : c'est la fonction qui prend deux petits tableaux triés en entrée et doit les fusionner pour produire un grand tableau trié, comme expliqué ci-dessus. Finalement, le « on recommence ! » est écrit par l'appel de la fonction `tri_fusion` elle-même au sein de sa propre définition : on appelle cela des *appels récursifs* ; la fonction elle-même s'appelle une *fonction récursive*. Nous reverrons plus loin ce mécanisme.

En terme de complexité, il n'est pas très difficile de se convaincre que la fusion de deux tableaux triés peut s'exécuter en temps linéaire ($O(n)$) en la taille du grand tableau. Par ailleurs, comme on peut le voir en FIGURE 1, on fait assez peu d'appels récursifs finalement : comme pour la recherche dichotomique, l'intérêt de la méthode est qu'à chaque appel récursif, on a divisé par deux la longueur du tableau à trier. On ne peut donc faire qu'au plus $O(\log_2 n)$ appels récursifs imbriqués. Ceci explique pourquoi la complexité du tri par fusion est en $O(n \log_2 n)$, ce qu'on admet dans ce cours.

On est donc passé d'un tri de complexité $O(n^2)$ à un tri de complexité $O(n \log_2 n)$: c'est évidemment bien mieux de la même façon que la recherche dichotomique de complexité $O(\log_2 n)$ est bien meilleure que la recherche séquentielle de complexité $O(n)$. Mais peut-on encore mieux faire ? En fait, on peut montrer qu'il n'est pas possible de mieux faire, dès lors qu'on ne considère que des tris qui procèdent par comparaison des éléments deux par deux, comme c'est le cas pour les tris qu'on a étudiés jusque-là.