Aix*Marseille université

PORTAIL ET L1EN2ANS RENÉ DESCARTES

Science informatique : Partiel 1h30

Année 2021-22

Documents et calculatrice interdits

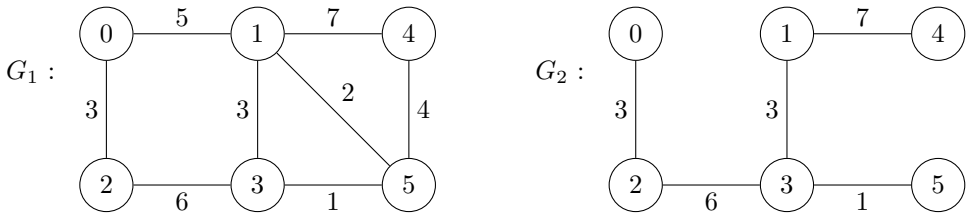
Tous les exercices sont indépendants et peuvent donc être traités dans n'importe quel ordre. Toute réponse non correctement rédigée (avec explication précise et concise) ou non justifiée sera considérée comme fausse.

Exercice 1 On considère le pseudo-code suivant d'une fonction prenant un tableau de caractères en argument.

```
fonction mystère(t : tableau de caractères):
    x := ['l','a','v','a']
    n := longueur(t)
    m := longueur(x)
    Pour i de 0 à n-m:
        j := 0
        Tant que j<m et t[i+j] = x[j]:
            j := j+1
        FinTantQue
        Si j = m alors:
            retourner(i)
        FinSi
    FinPour
    retourner(-1)
```

1. Exécuter la fonction `mystère` sur le tableau `['l','a',' ','l','a','v','a','n','d','e']` puis sur le tableau `['l','a','v','e','r']`, en détaillant les opérations effectuées et la valeur retournée.
2. Décrire en une phrase ce que retourne la fonction lorsqu'elle est appelée avec un tableau de caractères quelconque en entrée.
3. Justifier que la fonction `mystère` termine.
4. Donner, en la justifiant, la complexité dans le pire des cas de la fonction `mystère`, en fonction de la longueur n du tableau donné en entrée. On pourra simplifier le résultat en utilisant la notation « grand O ».

Exercice 2 Considérons des graphes non orientés avec des poids positifs sur les arêtes. Par exemple, un exemple de graphe G_1 est représenté à gauche ci-dessous :



Un problème important en pratique est celui de la recherche d'un *arbre couvrant de poids minimum*. Un arbre couvrant d'un graphe $G = (S, A)$ est un sous-ensemble $B \subseteq A$ d'arêtes tel que :

— le sous-graphe (S, B) obtenu en ne conservant que les arêtes de B est un graphe connexe ;

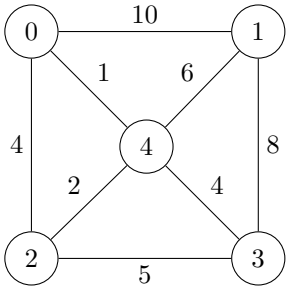
— le sous-graphe (S, B) est acyclique.

Un arbre couvrant est donc un sous-graphe de G qui permet de relier tous les sommets de G, sans pour autant créer de cycles. Par exemple, un arbre couvrant du graphe G_1 est dessiné à droite ci-dessus (c'est le graphe G_2).

Le poids d'un arbre couvrant est la somme des poids des arêtes qui le composent. Par exemple, l'arbre couvrant ci-dessus a poids $3+6+3+1+7 = 20$. Un *arbre couvrant de poids minimum* est un arbre couvrant qui est de plus petit poids parmi tous les arbres couvrants possibles.

1. Dessiner un arbre couvrant de poids minimum du graphe G_1 (on ne demande pas de justifier qu'il est de poids minimum).
2. Tous les graphes non orientés à poids positifs admettent-ils un arbre couvrant ?
3. Tous les graphes qui admettent un arbre couvrant admettent-ils un unique arbre couvrant de poids minimum ?

4. Il existe un algorithme très simple permettant de calculer un arbre couvrant de poids minimum lorsqu'il en existe un. Il s'agit de l'algorithme de Kruskal qui peut s'écrire ainsi :
   — On commence par trier les arêtes du graphe par ordre croissant de poids.
   — On crée un ensemble vide B d'arêtes.
   — Ensuite, on considère les arêtes $\{u, v\} \in A$ les unes après les autres dans cet ordre :
      — si l'ajout de l'arête $\{u, v\}$ dans B ne crée pas de cycle, alors on l'ajoute dans B.
   — On renvoie alors le sous-graphe (S, B).
   Exécuter l'algorithme de Kruskal sur le graphe G_3 ci-dessous, en détaillant les étapes intermédiaires.

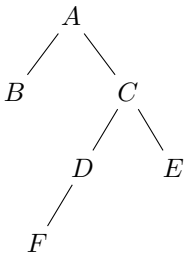


5. Un problème proche consiste à chercher un arbre couvrant de poids *maximum* (c'est-à-dire de poids maximum parmi tous les arbres couvrants possibles). Proposer une adaptation de l'algorithme de Kruskal pour pouvoir traiter ce cas-là. Illustrer ce nouvel algorithme sur le graphe G_1 du début de l'exercice.
6. Imaginer une situation du monde réel où la recherche d'arbre couvrant de poids minimum ou maximum permet de résoudre un problème pratique.

Exercice 3 Considérons la fonction suivante récursive écrite en pseudo-code, prenant un nœud d'arbre binaire en entrée :

```
fonction mystère(nœud) :
    Si nœud = nil alors
        retourner(0)
    Sinon
        x := mystère(enfant_gauche[nœud])
        y := mystère(enfant_droit[nœud])
        retourner(1+x+y)
    FinSi
```

1. Que renvoie cette fonction `mystère` lorsqu'on l'appelle sur le nœud A de l'arbre binaire suivant ? Vous expliquerez brièvement votre réponse.



2. Décrire en une phrase ce que retourne la fonction lorsqu'elle est appelée à la racine d'un arbre binaire quelconque.
3. Proposer le pseudo-code d'une fonction récursive qui retourne la hauteur d'un arbre binaire.

Exercice 4 Un palindrome est une chaîne de caractères dont la lecture est la même, qu'on lise de gauche à droite ou de droite à gauche. Par exemple, « ressasser » et « kayak » sont des palindromes.

1. Écrire en pseudo-code une fonction `palindrome` prenant en argument une chaîne de caractères et renvoyant `vrai` si cette chaîne de caractères est un palindrome et `faux` sinon. Comme dans le premier exercice, on représentera une chaîne de caractères sous la forme d'un tableau dont les cases successives contiennent les caractères composant la chaîne, dans l'ordre dans lequel ils apparaissent dans la chaîne. (Pensez à vérifier votre réponse au brouillon en simulant l'exécution de l'algorithme que vous avez proposé sur un ou deux petits exemples bien choisis.)
2. Question bonus : justifier que l'algorithme proposé est correct.

Aix*Marseille université

PORTAIL ET L1EN2ANS RENÉ DESCARTES

Science informatique : Partiel

Année 2021-22

Documents et calculatrice interdits

Tous les exercices sont indépendants et peuvent donc être traités dans n'importe quel ordre. Toute réponse non correctement rédigée (avec explication précise et concise) ou non justifiée sera considérée comme fausse.

Exercice 1 On considère le pseudo-code suivant d'une fonction prenant un tableau de caractères en argument.

```
fonction mystère(t : tableau de caractères):
    x := ['l','a','v','a']
    n := longueur(t)
    m := longueur(x)
    Pour i de 0 à n-m:
        j := 0
        Tant que j<m et t[i+j] = x[j]:
            j := j+1
        FinTantQue
        Si j = m alors:
            retourner(i)
        FinSi
    FinPour
    retourner(-1)
```

1. Exécuter la fonction `mystère` sur le tableau `['l','a',' ','l','a','v','a','n','d','e']` puis sur le tableau `['l','a','v','e','r']`, en détaillant les opérations effectuées et la valeur retournée.

Solution : On donne les valeurs successives prises par les variables `i` et `j` avant que la fonction ne s'arrête. Dans le premier cas, où `n` prend la valeur 10 et `m` la valeur 4, on obtient

i	j	t[i+j]	x[j]
0	0	'l'	'l'
0	1	'a'	'a'
0	2	' '	'v'
1	0	'a'	'l'
2	0	' '	'l'
3	0	'l'	'l'
3	1	'a'	'a'
3	2	'v'	'v'
3	3	'a'	'a'

On retourne donc la valeur 3.
Dans le second cas, où `n` prend la valeur 5 et `m` la valeur 4, on obtient

i	j	t[i+j]	x[j]
0	0	'l'	'l'
0	1	'a'	'a'
0	2	'v'	'v'
0	3	'e'	'a'
1	0	'a'	'l'

La boucle `Pour` s'arrête puisque $\texttt{n} - \texttt{m} = 1$. La fonction retourne donc la valeur -1.

2. Décrire en une phrase ce que retourne la fonction lorsqu'elle est appelée avec un tableau de caractères quelconque en entrée.

Solution : Si le tableau de caractères contient les caractères `'l'`, `'a'`, `'v'`, `'a'` dans des cases successives, alors la fonction retourne l'indice correspondant à la lettre `l` dans le tableau `t`, sinon la fonction retourne -1. Si la séquence est présente plusieurs fois, la fonction renvoie l'indice correspondant au `'l'` de la première instance rencontrée. La fonction recherche donc la première occurrence du motif *lava* dans un texte donné comme un tableau de caractères.

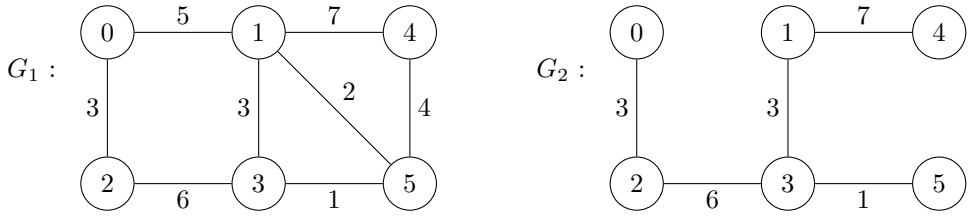
3. Justifier que la fonction `mystère` termine.

Solution : La seule possibilité pour que la fonction ne termine pas serait qu'elle reste indéfiniment dans la boucle `Tant que`. Si `j` devient supérieur ou égal à `m`, cette boucle se termine puisqu'alors le test `j < m et t[i + j] = x[j]` n'est plus vérifié. Or, `j` est un entier positif, qui augmente strictement lors de chaque itération de la boucle et `m` est un entier fixé. On a, par conséquent, forcément `j` $\geq$ `m` après un nombre fini d'itérations. La boucle `Tant que`, et donc la fonction entière, terminent donc toujours.

4. Donner, en la justifiant, la complexité dans le pire des cas de la fonction `mystère`, en fonction de la longueur n du tableau donné en entrée. On pourra simplifier le résultat en utilisant la notation « grand O ».

Solution : Chaque itération de la boucle `Tant que`, y compris le test en début d'itération, effectue un nombre borné d'opérations élémentaires, disons au plus 5. On peut affiner le raisonnement de la question précédente en remarquant que la variable `j` part de 0 et est incrémentée de exactement 1 à chaque itération de la boucle `Tant que`, qui réalise donc au plus `m` itérations. Dans cette fonction `m` est constant et égal à 4, la longueur du tableau `x`. Au total, chaque itération de la boucle `Pour` coûte donc $1 + 4 \times 5 = 21$ opérations élémentaires. Au total, la boucle `Pour` effectue donc $(n - 4 + 1) \times 21$ opérations élémentaires. Lorsqu'on ajoute les 4 opérations élémentaires en dehors de la boucle, on obtient une complexité de l'ordre de $21n + 67$ opérations élémentaires. Il s'agit d'une complexité linéaire, c'est-à-dire en $\mathcal{O}(n)$.

Exercice 2 Considérons des graphes non orientés avec des poids positifs sur les arêtes. Par exemple, un exemple de graphe G_1 est représenté à gauche ci-dessous :



Un problème important en pratique est celui de la recherche d'un *arbre couvrant de poids minimum*. Un arbre couvrant d'un graphe $G = (S, A)$ est un sous-ensemble $B \subseteq A$ d'arêtes tel que :

— le sous-graphe (S, B) obtenu en ne conservant que les arêtes de B est un graphe connexe ;

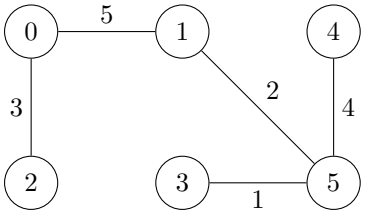
— le sous-graphe (S, B) est acyclique.

Un arbre couvrant est donc un sous-graphe de G qui permet de relier tous les sommets de G, sans pour autant créer de cycles. Par exemple, un arbre couvrant du graphe G_1 est dessiné à droite ci-dessus (c'est le graphe G_2).

Le poids d'un arbre couvrant est la somme des poids des arêtes qui le composent. Par exemple, l'arbre couvrant ci-dessus a poids $3+6+3+1+7 = 20$. Un *arbre couvrant de poids minimum* est un arbre couvrant qui est de plus petit poids parmi tous les arbres couvrants possibles.

1. Dessiner un arbre couvrant de poids minimum du graphe G_1 (on ne demande pas de justifier qu'il est de poids minimum).

Solution : L'unique arbre couvrant de poids minimum de G_1 est l'arbre suivant de poids 15 :

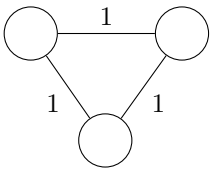


2. Tous les graphes non orientés à poids positifs admettent-ils un arbre couvrant ?

Solution : Non, un graphe non connexe n'admet pas d'arbre couvrant.

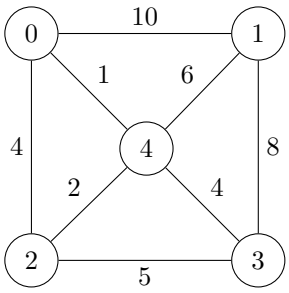
3. Tous les graphes qui admettent un arbre couvrant admettent-ils un unique arbre couvrant de poids minimum ?

Solution : Non, par exemple le graphe suivant admet trois arbres couvrants de poids minimum 2 (il faut supprimer l'une quelconque des trois arêtes) :



4. Il existe un algorithme très simple permettant de calculer un arbre couvrant de poids minimum lorsqu'il en existe un. Il s'agit de l'algorithme de Kruskal qui peut s'écrire ainsi :
   — On commence par trier les arêtes du graphe par ordre croissant de poids.
   — On crée un ensemble vide B d'arêtes.
   — Ensuite, on considère les arêtes $\{u, v\} \in A$ les unes après les autres dans cet ordre :
      — si l'ajout de l'arête $\{u, v\}$ dans B ne crée pas de cycle, alors on l'ajoute dans B.
   — On renvoie alors le sous-graphe (S, B).

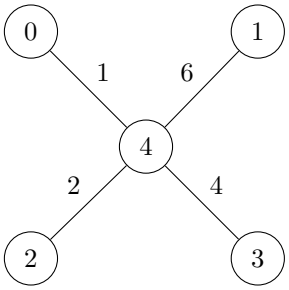
   Exécuter l'algorithme de Kruskal sur le graphe G_3 ci-dessous, en détaillant les étapes intermédiaires.



Solution : On considère les arêtes de G_3 dans l'ordre croissant de leur poids :

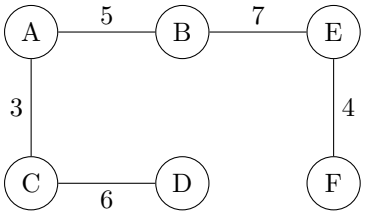
(a) l'arête $\{0, 4\}$ ne crée pas de cycle, on l'ajoute à B ;
(b) l'arête $\{2, 4\}$ ne crée pas de cycle, on l'ajoute à B ;
(c) l'arête $\{0, 2\}$ crée le cycle $(0, 2, 4)$, donc on ne l'ajoute pas à B ;
(d) l'arête $\{3, 4\}$ ne crée pas de cycle, on l'ajoute à B ;
(e) l'arête $\{2, 3\}$ crée le cycle $(2, 3, 4)$, donc on ne l'ajoute pas à B ;
(f) l'arête $\{1, 4\}$ ne crée pas de cycle, on l'ajoute à B ;
(g) l'arête $\{1, 3\}$ crée un cycle, on ne l'ajoute pas à B ;
(h) l'arête $\{0, 1\}$ crée un cycle, on ne l'ajoute pas à B.

On renvoie donc l'ensemble $B = \big\{\{0, 4\}, \{2, 4\}, \{3, 4\}, \{1, 4\}\big\}$, c'est-à-dire l'arbre couvrant



5. Un problème proche consiste à chercher un arbre couvrant de poids *maximum* (c'est-à-dire de poids maximum parmi tous les arbres couvrants possibles). Proposer une adaptation de l'algorithme de Kruskal pour pouvoir traiter ce cas-là. Illustrer ce nouvel algorithme sur le graphe G_1 du début de l'exercice.

Solution : Il suffit de parcourir désormais les arêtes du graphe par ordre *décroissant* de poids. Sur l'exemple du graphe G_1, on obtient alors l'arbre couvrant suivant de poids 25 :



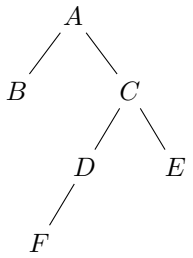
6. Imaginer une situation du monde réel où la recherche d'arbre couvrant de poids minimum ou maximum permet de résoudre un problème pratique.

Solution : On peut imaginer un graphe représentant un réseau de fibre optique dans une ville : les sommets représentent les immeubles à câbler, les arêtes représentent les rues qui relient les immeubles et les poids sur les arêtes représentent la distance entre les deux immeubles. On se place dans la peau d'un gestionnaire de réseau téléphonique qui doit relier l'ensemble des immeubles avec la fibre optique, mais qui aimerait le faire en ayant le moins possible de câble à dérouler. Cela revient à chercher un arbre couvrant de poids minimum.
Imaginons désormais une autre situation où un gestionnaire de réseau routier s'apprête à mettre en place des péages sur certaines routes et à ne plus maintenir les routes sans péages. Pour conserver un réseau routier viable et limiter les travaux de rénovation des routes, il cherche également à construire un réseau en forme d'arbre couvrant. Par contre, son objectif est de maximiser les revenus dus aux péages, ceux-ci étant proportionnels à la longueur des tronçons de routes. Il modélise donc la situation avec un graphe non orienté à poids positifs, où le poids d'une arête représente les droits de péage de la route.

Exercice 3 Considérons la fonction suivante récursive écrite en pseudo-code, prenant un nœud d'arbre binaire en entrée :

```
fonction mystère(nœud) :
    Si nœud = nil alors
        retourner(0)
    Sinon
        x := mystère(enfant_gauche[nœud])
        y := mystère(enfant_droit[nœud])
        retourner(1+x+y)
    FinSi
```

1. Que renvoie cette fonction `mystère` lorsqu'on l'appelle sur le nœud A de l'arbre binaire suivant ? Vous expliquerez brièvement votre réponse.



Solution : Le nœud A n'est pas égal à `nil` donc
- on réalise l'appel récursif sur le nœud B, qui n'est pas égal à `nil` donc
  - on réalise l'appel récursif sur son enfant gauche, qui est vide et qui retourne donc 0
  - puis on réalise l'appel récursif sur son enfant droit, qui est vide et qui retourne donc 0
  - donc on retourne $1 + 0 + 0 = 1$.
- on réalise alors l'appel récursif sur l'enfant droit de A, c'est-à-dire le nœud C, qui n'est pas égal à `nil` donc
  - on réalise l'appel récursif sur son enfant gauche, le nœud D, qui n'est pas vide donc
    - on réalise l'appel récursif sur son enfant gauche, le nœud F qui va aussi réaliser deux appels récursifs sur des arbres `nil` et retourner 1
    - on réalise ensuite l'appel récursif sur l'enfant droit de D, qui est vide ce qui retourne 0
    - on retourne alors $1 + 1 + 0 = 2$
  - on réalise ensuite l'appel récursif sur l'enfant droit de C, le nœud E qui a deux enfants vides et qui retourne donc 1
  - on retourne alors $1 + 2 + 1 = 4$
- à la racine, on retourne donc $1 + 1 + 4 = 6$.

On retourne donc la valeur 6 à l'appel initial.

2. Décrire en une phrase ce que retourne la fonction lorsqu'elle est appelée à la racine d'un arbre binaire quelconque.

Solution : La fonction `mystère` retourne la taille (nombre de nœuds) de l'arbre binaire fourni en entrée.

3. Proposer le pseudo-code d'une fonction récursive qui retourne la hauteur d'un arbre binaire.

Solution : On suit la même idée mais on doit adapter la ligne 3 du code, puisque la hauteur d'un arbre vide vaut -1 et non 0, et la ligne 7 puisque la hauteur d'un arbre non vide vaut la somme de 1 et du maximum de la hauteur de ses deux enfants (notons que cela marche aussi pour trouver que la hauteur d'une feuille vaut $0 = 1 + \max(-1, -1)$).

```
fonction hauteur(nœud) :
    Si nœud ≠ nil alors
        retourner(-1)
    Sinon
        x := mystère(enfant_gauche[nœud])
        y := mystère(enfant_droit[nœud])
        retourner(1+max(x,y))
    FinSi
```

Notons que, si on ne s'autorise pas l'utilisation d'une fonction `max`, on peut remplacer l'avant dernière-ligne par

```
    Si x>y alors:
        retourner(1+x)
    Sinon:
        retourner(1+y)
    FinSi
```

Exercice 4 Un palindrome est une chaîne de caractères dont la lecture est la même, qu'on lise de gauche à droite ou de droite à gauche. Par exemple, « ressasser » et « kayak » sont des palindromes.

1. Écrire en pseudo-code une fonction `palindrome` prenant en argument une chaîne de caractères et renvoyant `vrai` si cette chaîne de caractères est un palindrome et `faux` sinon. Comme dans le premier exercice, on représentera une chaîne de caractères sous la forme d'un tableau dont les cases successives contiennent les caractères composant la chaîne, dans l'ordre dans lequel ils apparaissent dans la chaîne. (Pensez à vérifier votre réponse au brouillon en simulant l'exécution de l'algorithme que vous avez proposé sur un ou deux petits exemples bien choisis.)

Solution : Plusieurs solutions sont possibles. Par exemple, on peut créer un tableau dans lequel on met une copie « inversée » de la chaîne donnée en entrée, et on compare ensuite la chaîne originale et sa version inversée case par case.

```
fonction palindrome(t: tableau de caractères):
    n := longueur(t)
    x := tableau de longueur n
    Pour i de 0 à n-1:
        x[i] := t[n-i]
    FinPour
    est_un_palindrome := vrai
    i := 0
    Tant que est_un_palindrome et i < n:
        Si t[i] ≠ x[i]:
            est_un_palindrome := faux
        FinSi
        i := i+1
    FinTantQue
    retourner(est_un_palindrome)
```

Une solution plus économe en mémoire consiste à parcourir le tableau initial dans les deux sens simultanément (de gauche à droite et de droite à gauche) ce qui évite d'avoir à faire une copie.

```
fonction palindrome_v2(t: tableau de caractères):
    n := longueur(t)
    est_un_palindrome := vrai
    i := 0
    Tant que est_un_palindrome et i < n:
        Si t[i] ≠ t[n-1-i]:
            est_un_palindrome := faux
        FinSi
        i := i+1
    FinTantQue
    retourner(est_un_palindrome)
```

On peut encore améliorer cette version, pour économiser des cycles de processeur cette fois, en remarquant qu'elle fait chaque vérification en double. Par exemple, pour l'entrée « kayak », de longueur 5, une fois qu'on a vérifié que le premier caractère et le dernier caractère sont identiques ('k') et que le second caractère et l'avant-dernier sont identiques ('a'), on peut s'arrêter. En général on peut s'arrêter après au plus $\lfloor n/2 \rfloor$ itérations, où n est la longueur de la chaîne de caractères donnée en entrée.

```
fonction palindrome_v3(t: tableau de caractères):
    n := longueur(t)
    est_un_palindrome := vrai
    i := 0
    Tant que est_un_palindrome et i < ⌊n/2⌋:
        Si t[i] ≠ t[n-1-i]:
            est_un_palindrome := faux
        FinSi
        i := i+1
    FinTantQue
    retourner(est_un_palindrome)
```

2. Question bonus : justifier que l'algorithme proposé est correct.

Solution : Justifions la correction de `palindrome_v2`. La preuve de correction pour la première version est similaire et nous reviendrons sur la preuve de correction de `palindrome_v3` à la fin.

- Commençons par réécrire la définition d'un palindrome dans un language plus formel. Par définition, une chaîne de caractères t de longueur n est un palindrome si et seulement si $t[i-1]$, le i-ième caractère en partant de la gauche est identique à $t[n-i]$, le i-ième caractère en partant de la droite, pour tout entier i entre 1 et n (inclus).
- Notons $P(k)$ la proposition : $t[i-1]$, le i-ième caractère en partant de la gauche est identique à $t[n-i]$, le i-ième caractère en partant de la droite, pour tout entier i entre 1 et k (inclus). Remarquez que nous venons de voir que t est un palindrome si et seulement si $P(n)$ est vraie.
- On montre facilement par récurrence que `palindrome_v2` possède l'invariant de boucle suivant : après k itérations de la boucle `Tant que`, `est_un_palindrome` prend la valeur 1 si $P(k)$ est vraie et 0 sinon. (Notez que $P(0)$ est toujours vraie, c'est à dire qu'une chaîne de caractères vide est un palindrome).
- Pour pouvoir conclure notre raisonnement, nous avons aussi besoin de montrer que l'algorithme termine. La seule possibilité pour que ce ne soit pas le cas, serait de rester coincé dans la boucle `Tant que`. Cela ne peut arriver, car `i` est un entier initialisé à 0 et incrémenté de 1 à chaque itération, de sorte que la condition de terminaison de la boucle (`est_un_palindrome et i<n`) sera vérifiée après au plus n itérations. (Ce dont on peut déduire, au passage, que l'algorithme a une complexité dans le pire cas en $O(n)$.)
- Nous sommes à présent en mesure de conclure. Nous savons que l'algorithme quitte toujours la boucle `Tant que` car nous avons prouvé que l'algorithme termine. La valeur de `est_un_palindrome` renvoyée par l'algorithme est celle obtenue juste après la dernière itération de cette boucle. À ce moment de l'exécution, la condition de terminaison de la boucle vient d'être réalisée, ce qui signifie que soit `est_un_palindrome` vaut 0 et `i` est inférieur ou égal à `n`, soit `est_un_palindrome` vaut 1 et `i` est égal à `n` (`i` étant incrémenté de un en un il ne pourra pas être strictement supérieur à `n`).
  - Dans le premier cas, $\texttt{i} \leq \texttt{n}$ et d'après notre invariant de boucle $P(\texttt{i})$ est fausse. Or, si $P(a)$ est fausse pour $a \leq b$, alors $P(b)$ est fausse. Donc $P(\texttt{n})$ est fausse, c'est à dire que l'entrée n'est pas un palindrome et la valeur renvoyée (`est_un_palindrome` = `faux`) est correcte.
  - Dans le deuxième cas, d'après notre invariant de boucle $P(\texttt{n})$ est vraie et donc l'entrée est un palindrome et la valeur renvoyée (`est_un_palindrome` = `vrai`) est correcte.

  Pour prouver la correction de `palindrome_v3`, il suffit de reprendre le raisonnement utilisé pour `palindrome_v2` en remarquant que si $P(\lfloor \texttt{n}/2 \rfloor)$ est vraie alors $P(\texttt{n})$ est vraie.