Algorithmique

Analyse de quelques algorithmes

LYCÉE CARNOT (DIJON), 2016 - 2017

Sommaire

Sommaire

Factorielle $n!$

entrée: n un entier naturel
résultat: $n!$
Factorielle(n)

```
1: prod←1
2: pour i entre 2 et n faire
3:     prod ←prod×i
4: fin pour
5: renvoi: prod
```

Factorielle $n!$

Invariant de boucle :

entrée: n un entier naturel
résultat: $n!$
Factorielle(n)

```
1: prod←1
2: pour i entre 2 et n faire
3:     prod ←prod×i
4: fin pour
5: renvoi: prod
```

Factorielle $n!$

entrée: n un entier naturel
résultat: $n!$
Factorielle(n)

```
1: prod←1
2: pour i entre 2 et n faire
3:     prod ←prod×i
4: fin pour
5: renvoi: prod
```

Invariant de boucle :

`prod` contient $i!$ à chaque fin d'itération

Factorielle *n*!

entrée: *n* un entier naturel
résultat: *n*!
Factorielle(n)

```
1: prod←1
2: pour i entre 2 et n faire
3:     prod ←prod×i
4: fin pour
5: renvoi: prod
```

Invariant de boucle :

`prod` contient *i*! à chaque fin d'itération

Complexité :

Factorielle $n!$

entrée: n un entier naturel
résultat: $n!$
Factorielle(n)

```
1: prod←1
2: pour i entre 2 et n faire
3:     prod ←prod×i
4: fin pour
5: renvoi: prod
```

Invariant de boucle :

`prod` contient $i!$ à chaque fin d'itération

Complexité :

$n - 1$ multiplications et n affectations, soit $2.n - 1$ opérations.

Factorielle $n!$

entrée: n un entier naturel
résultat: $n!$
Factorielle(n)

```
1: prod←1
2: pour i entre 2 et n faire
3:     prod ←prod×i
4: fin pour
5: renvoi: prod
```

Invariant de boucle :

`prod` contient $i!$ à chaque fin d'itération

Complexité :

$n-1$ multiplications et n affectations, soit $2.n-1$ opérations.

$$C(n) = \Theta(n)$$

Factorielle $n!$

entrée: n un entier naturel
résultat: $n!$
Factorielle(n)

```
1: prod←1
2: pour i entre 2 et n faire
3:     prod ←prod×i
4: fin pour
5: renvoi: prod
```

Invariant de boucle :
`prod` contient $i!$ à chaque fin d'itération

Complexité :
$n-1$ multiplications et n affectations, soit $2.n-1$ opérations.

$$C(n) = \Theta(n)$$

Terminaison ? :

Factorielle $n!$

entrée: n un entier naturel
résultat: $n!$
Factorielle(n)
1: prod←1
2: **pour** i entre 2 et n **faire**
3: prod ←prod×i
4: **fin pour**
5: **renvoi:** prod

Invariant de boucle :
`prod` contient $i!$ à chaque fin d'itération

Complexité :
$n-1$ multiplications et n affectations, soit $2.n-1$ opérations.

$$C(n) = \Theta(n)$$

Terminaison ? :
La question ne se pose pas ici !

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Invariant de boucle :

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Invariant de boucle :
`puiss` contient x^i à chaque fin d'itération

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Invariant de boucle :
`puiss` contient x^i à chaque fin d'itération

Complexité :

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Invariant de boucle :
`puiss` contient x^i à chaque fin d'itération

Complexité :
n multiplications et $n+1$ affectations, soit $2.n+1$ opérations.

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Invariant de boucle :
`puiss` contient x^i à chaque fin d'itération

Complexité :
n multiplications et $n+1$ affectations, soit $2.n+1$ opérations.

$$C(n) = \Theta(n)$$

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Invariant de boucle :
`puiss` contient x^i à chaque fin d'itération

Complexité :
n multiplications et $n+1$ affectations, soit $2.n+1$ opérations.

$$C(n) = \Theta(n)$$

Terminaison ? :

Puissance x^n

entrée: x un nombre réel, n un entier naturel
résultat: x^n
Puissance(x,n)

```
1: puiss←1
2: pour i entre 1 et n faire
3:     puiss ←puiss×x
4: fin pour
5: renvoi: puiss
```

Invariant de boucle :
`puiss` contient x^i à chaque fin d'itération

Complexité :
n multiplications et $n+1$ affectations, soit $2.n+1$ opérations.

$$C(n) = \Theta(n)$$

Terminaison ? :
La question ne se pose pas ici !

Somme(L)

entrée: *L* un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

Somme(L)

```
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Somme(L)

Invariant de boucle :

entrée: *L* un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

```
Somme(L)
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Somme(L)

entrée: L un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

Somme(L)

```
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Invariant de boucle :

`som` contient $i + 1$ premiers éléments de L à chaque fin d'itération.

Somme(L)

entrée: *L* un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

Somme(L)

```
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Invariant de boucle :

`som` contient $i + 1$ premiers éléments de *L* à chaque fin d'itération.

Complexité :

Somme(L)

entrée: L un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

Somme(L)

```
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Invariant de boucle :

`som` contient $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Si n est la taille de L, n additions, n accès à L et $n+2$ affectations, soit $3.n+2$ opérations élémentaires.

Somme(L)

entrée: L un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

Somme(L)

```
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Invariant de boucle :

`som` contient $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Si n est la taille de L, n additions, n accès à L et $n+2$ affectations, soit $3.n+2$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Somme(L)

entrée: L un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

Somme(L)

```
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Invariant de boucle :

`som` contient $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Si n est la taille de L, n additions, n accès à L et $n+2$ affectations, soit $3.n+2$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Terminaison ? :

Somme(L)

entrée: L un tableau de nombres réels
résultat: *som* la somme des éléments du tableau

Somme(L)

```
1: som←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     som ←som+L[i]
5: fin pour
6: renvoi: som
```

Invariant de boucle :

`som` contient $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Si n est la taille de L, n additions, n accès à L et $n+2$ affectations, soit $3.n+2$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Terminaison ? :

La question ne se pose pas ici !

Maximum(L)

entrée: *L* un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

Maximum(L)

```
1: n←taille(L)
2: si n==0 alors
3:     renvoi: Tableau vide !
4: fin si
5: max←L[0]
6: pour i entre 1 et n-1 faire
7:     si L[i]>max alors
8:         max ←L[i]
9:     fin si
10: fin pour
11: renvoi: max
```

Maximum(L)

entrée: *L* un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

```
Maximum(L)
1:  n←taille(L)
2:  si n==0 alors
3:      renvoi: Tableau vide !
4:  fin si
5:  max←L[0]
6:  pour i entre 1 et n-1 faire
7:      si L[i]>max alors
8:          max ←L[i]
9:      fin si
10: fin pour
11: renvoi: max
```

Invariant de boucle :

Maximum(L)

entrée: L un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

```
Maximum(L)
1:  n←taille(L)
2:  si n==0 alors
3:      renvoi: Tableau vide !
4:  fin si
5:  max←L[0]
6:  pour i entre 1 et n-1 faire
7:      si L[i]>max alors
8:          max ←L[i]
9:      fin si
10: fin pour
11: renvoi: max
```

Invariant de boucle :

`max` contient le maximum des $i+1$ premiers éléments de L à chaque fin d'itération.

Maximum(L)

entrée: L un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

```
Maximum(L)
1:  n←taille(L)
2:  si n==0 alors
3:      renvoi: Tableau vide !
4:  fin si
5:  max←L[0]
6:  pour i entre 1 et n-1 faire
7:      si L[i]>max alors
8:          max ←L[i]
9:      fin si
10: fin pour
11: renvoi: max
```

Invariant de boucle :

`max` contient le maximum des $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Maximum(L)

entrée: L un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

```
Maximum(L)
1:  n←taille(L)
2:  si n==0 alors
3:      renvoi: Tableau vide !
4:  fin si
5:  max←L[0]
6:  pour i entre 1 et n-1 faire
7:      si L[i]>max alors
8:          max ←L[i]
9:      fin si
10: fin pour
11: renvoi: max
```

Invariant de boucle :

`max` contient le maximum des $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Si n est la taille de L, $n-1$ comparaison, au pire $n-1$ accès à L et $n-1$ affectations pour la boucle, soit au pire des cas $3.n-1$ opérations élémentaires.

Maximum(L)

entrée: L un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

```
Maximum(L)
1:  n←taille(L)
2:  si n==0 alors
3:      renvoi: Tableau vide !
4:  fin si
5:  max←L[0]
6:  pour i entre 1 et n-1 faire
7:      si L[i]>max alors
8:          max ←L[i]
9:      fin si
10: fin pour
11: renvoi: max
```

Invariant de boucle :

`max` contient le maximum des $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Si n est la taille de L, $n-1$ comparaison, au pire $n-1$ accès à L et $n-1$ affectations pour la boucle, soit au pire des cas $3.n-1$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Maximum(L)

entrée: L un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

```
Maximum(L)
1: n←taille(L)
2: si n==0 alors
3:     renvoi: Tableau vide !
4: fin si
5: max←L[0]
6: pour i entre 1 et n-1 faire
7:     si L[i]>max alors
8:         max ←L[i]
9:     fin si
10: fin pour
11: renvoi: max
```

Invariant de boucle :

`max` contient le maximum des $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

Si n est la taille de L, $n-1$ comparaison, au pire $n-1$ accès à L et $n-1$ affectations pour la boucle, soit au pire des cas $3.n-1$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Terminaison ? :

Maximum(L)

entrée: L un tableau de nombres réels
résultat: *max* le plus grand élément du tableau

```
Maximum(L)
1:  n←taille(L)
2:  si n==0 alors
3:      renvoi: Tableau vide !
4:  fin si
5:  max←L[0]
6:  pour i entre 1 et n-1 faire
7:      si L[i]>max alors
8:          max ←L[i]
9:      fin si
10: fin pour
11: renvoi: max
```

Invariant de boucle :
`max` contient le maximum des $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :
Si n est la taille de L, $n-1$ comparaison, au pire $n-1$ accès à L et $n-1$ affectations pour la boucle, soit au pire des cas $3.n-1$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Terminaison ? :
Boucle inconditionnelle, donc pas de problème.

Occurence(x,L)

entrée: *x* un nombre réel et *L* un tableau de nombres réels
résultat: *nb* le nombre d'occurences de *x* dans *L*

Occurence(x,L)

```
1: nb←0
2: n←taille(L)
3: pour i entre 0 et n-1 faire
4:     si L[i]=x alors
5:         nb ←nb+1
6:     fin si
7: fin pour
8: renvoi: nb
```

Invariant de boucle :

Invariant de boucle :

`nb` contient le nombre d'occurrences de x parmi les $i+1$ premiers éléments de L à chaque fin d'itération.

Invariant de boucle :

`nb` contient le nombre d'occurrences de x parmi les $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

- **au mieux :** x n'apparaît jamais : n tests, 2 affectations soit $n+2$ opérations
- **au pire :** la liste ne contient que des x : n tests, n additions et $n+2$ affectations soit $3.n+2$ opérations

Invariant de boucle :

`nb` contient le nombre d'occurrences de x parmi les $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

- **au mieux :** x n'apparaît jamais : n tests, 2 affectations soit $n+2$ opérations
- **au pire :** la liste ne contient que des x : n tests, n additions et $n+2$ affectations soit $3.n+2$ opérations

$$C(n) = \Theta(n)$$

Invariant de boucle :

`nb` contient le nombre d'occurrences de x parmi les $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

- **au mieux :** x n'apparaît jamais : n tests, 2 affectations soit $n+2$ opérations
- **au pire :** la liste ne contient que des x : n tests, n additions et $n+2$ affectations soit $3.n+2$ opérations

$$C(n) = \Theta(n)$$

Terminaison ? :

Invariant de boucle :
`nb` contient le nombre d'occurrences de x parmi les $i+1$ premiers éléments de L à chaque fin d'itération.

Complexité :

- **au mieux :** x n'apparaît jamais : n tests, 2 affectations soit $n+2$ opérations
- **au pire :** la liste ne contient que des x : n tests, n additions et $n+2$ affectations soit $3.n+2$ opérations

$$C(n) = \Theta(n)$$

Terminaison ? :
Boucle inconditionnelle, donc pas de problème.

Miroir(chaine)

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Miroir(chaine)

Invariant de boucle :

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Miroir(chaine)

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Invariant de boucle :
`mir` contient les *i* derniers caractères de `chaine` dans l'ordre inverse à chaque fin d'itération.

Miroir(chaine)

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Invariant de boucle :
`mir` contient les *i* derniers caractères de `chaine` dans l'ordre inverse à chaque fin d'itération.

Complexité :

Miroir(chaine)

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Invariant de boucle :
`mir` contient les i derniers caractères de `chaine` dans l'ordre inverse à chaque fin d'itération.

Complexité :
Si n est la taille de `chaine`, on compte $n + 2$ affectations, n concaténations en bout de chaîne, n recherches de caractères de `chaine` soit $3.n + 2$ opérations élémentaires.

Miroir(chaine)

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Invariant de boucle :

`mir` contient les i derniers caractères de `chaine` dans l'ordre inverse à chaque fin d'itération.

Complexité :

Si n est la taille de `chaine`, on compte $n+2$ affectations, n concaténations en bout de chaîne, n recherches de caractères de `chaine` soit $3.n+2$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Miroir(chaine)

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Invariant de boucle :
`mir` contient les i derniers caractères de `chaine` dans l'ordre inverse à chaque fin d'itération.

Complexité :
Si n est la taille de `chaine`, on compte $n+2$ affectations, n concaténations en bout de chaîne, n recherches de caractères de `chaine` soit $3.n+2$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Terminaison ? :

Miroir(chaine)

entrée: *chaine* une chaîne de caractères
résultat: *eniahc* chaîne de caractères miroir de *chaine*
Miroir(chaine)

```
1: n←taille(L)
2: mir←""
3: pour i entre 1 et n faire
4:     mir←mir+chaine(n-1)
5: fin pour
6: renvoi: mir
```

Invariant de boucle :
`mir` contient les i derniers caractères de `chaine` dans l'ordre inverse à chaque fin d'itération.

Complexité :
Si n est la taille de `chaine`, on compte $n + 2$ affectations, n concaténations en bout de chaîne, n recherches de caractères de `chaine` soit $3.n + 2$ opérations élémentaires.

$$C(n) = \Theta(n)$$

Terminaison ? :
Boucle inconditionnelle, donc pas

Sommaire

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$

PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Plus petite puissance de 2 majorante

Invariant de boucle :

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$

PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$

PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Invariant de boucle :
`PpP` contient 2^i à la fin de chaque itération i.

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$

PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Invariant de boucle :
`PpP` contient 2^i à la fin de chaque itération i.

Terminaison :

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$
PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Invariant de boucle :
`PpP` contient 2^i à la fin de chaque itération i.

Terminaison :
La suite entière 2^i est strictement croissante et non majorée. Elle finit donc par dépasser n.

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: *PpP* le plus petite puissance de 2 telle que $n \leq PpP$

PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Invariant de boucle :
`PpP` contient 2^i à la fin de chaque itération i.

Terminaison :
La suite entière 2^i est strictement croissante et non majorée. Elle finit donc par dépasser n.

Complexité :

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$
PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Invariant de boucle :
`PpP` contient 2^i à la fin de chaque itération i.

Terminaison :
La suite entière 2^i est strictement croissante et non majorée. Elle finit donc par dépasser n.

Complexité :

$$n \leq 2^i \quad \Rightarrow \quad \log_2(n) \leq i$$

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$

PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Invariant de boucle :
`PpP` contient 2^i à la fin de chaque itération i.

Terminaison :
La suite entière 2^i est strictement croissante et non majorée. Elle finit donc par dépasser n.

Complexité :

$$n \leq 2^i \quad \Rightarrow \quad \log_2(n) \leq i$$
$$\Rightarrow \quad i \leq \lceil \log_2(n) \rceil$$

Plus petite puissance de 2 majorante

entrée: n un entier
résultat: PpP le plus petite puissance de 2 telle que $n \leq PpP$
PpPuissdedeux(n)

```
1: PpP←1
2: tant que PpP<n faire
3:     PpP←2*PpP
4: fin tant que
5: renvoi: PpP
```

Invariant de boucle :
`PpP` contient 2^i à la fin de chaque itération i.

Terminaison :
La suite entière 2^i est strictement croissante et non majorée. Elle finit donc par dépasser n.

Complexité :

$$\begin{array}{rcl} n \leq 2^i & \Rightarrow & \log_2(n) \leq i \\ & \Rightarrow & i \leq \lceil \log_2(n) \rceil \\ C(n) & = & \Theta(\ln(n)) \end{array}$$

Palindrome

entrée: *chaine* une chaîne de caractères
résultat: `vrai` si *chaine* est un palindrome ; `faux` sinon
Palindrome(chaine)

```
1: n←taille(chaine)
2: i←0
3: tant que i < ⌊n/2⌋ et chaine[i] = chaine[n – 1 – i] faire
4:     i←i+1
5: fin tant que
6: renvoi: i==n//2
```

Invariant de boucle :

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Terminaison :

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Terminaison :

La suite entière i est strictement croissante et non majorée. Elle finit donc par dépasser $\left\lfloor \frac{n}{2} \right\rfloor$.

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Terminaison :

La suite entière i est strictement croissante et non majorée. Elle finit donc par dépasser $\left\lfloor \frac{n}{2} \right\rfloor$.

Si `chaine` n'est pas un palindrome, l'algorithme s'arrête avant.

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Terminaison :

La suite entière i est strictement croissante et non majorée. Elle finit donc par dépasser $\left\lfloor \frac{n}{2} \right\rfloor$.

Si `chaine` n'est pas un palindrome, l'algorithme s'arrête avant.

Complexité en comparaison :

Au mieux 1 si la chaîne de caractères est vide.

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Terminaison :

La suite entière i est strictement croissante et non majorée. Elle finit donc par dépasser $\left\lfloor \frac{n}{2} \right\rfloor$.

Si `chaine` n'est pas un palindrome, l'algorithme s'arrête avant.

Complexité en comparaison :

Au mieux 1 si la chaîne de caractères est vide. Si la première et dernière lettre sont différentes, le coût est de 3.

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Terminaison :

La suite entière i est strictement croissante et non majorée. Elle finit donc par dépasser $\left\lfloor \frac{n}{2} \right\rfloor$.

Si `chaine` n'est pas un palindrome, l'algorithme s'arrête avant.

Complexité en comparaison :

Au mieux 1 si la chaîne de caractères est vide. Si la première et dernière lettre sont différentes, le coût est de 3.

Dans le cas d'un vrai palindrome $2.\left\lfloor \frac{n}{2} \right\rfloor + 2$.

Invariant de boucle :

Les $i+1$ premières et dernières lettres de `chaine` sont égales à la fin de chaque itération i.

Terminaison :

La suite entière i est strictement croissante et non majorée. Elle finit donc par dépasser $\left\lfloor \frac{n}{2} \right\rfloor$.

Si `chaine` n'est pas un palindrome, l'algorithme s'arrête avant.

Complexité en comparaison :

Au mieux 1 si la chaîne de caractères est vide. Si la première et dernière lettre sont différentes, le coût est de 3.

Dans le cas d'un vrai palindrome $2.\left\lfloor \frac{n}{2} \right\rfloor + 2$.

$$C(n) = O(n)$$

Test si *n* est une factorielle

entrée: *n* un entier
résultat: `vrai` si *n* est une factorielle ; `faux` sinon
IsFact(n)

```
1: fact←1
2: i←0
3: tant que fact < n faire
4:     i←i+1
5:     fract←fact*i
6: fin tant que
7: renvoi: fact=n
```

Invariant de boucle :

Invariant de boucle :

`Fac` contient $i!$ à la fin de chaque itération.

Invariant de boucle :

`Fac` contient $i!$ à la fin de chaque itération.

Terminaison :

Invariant de boucle :
`Fac` contient $i!$ à la fin de chaque itération.

Terminaison :
La suite entière $i!$ est strictement croissante et non majorée. Elle finit donc par dépasser n.

Invariant de boucle :
`Fac` contient $i!$ à la fin de chaque itération.

Terminaison :
La suite entière $i!$ est strictement croissante et non majorée. Elle finit donc par dépasser n.

Correction :
On sort de la boucle pour la plus petite valeur de i telle que $i! \geq n$.

Invariant de boucle :
`Fac` contient $i!$ à la fin de chaque itération.

Terminaison :
La suite entière $i!$ est strictement croissante et non majorée. Elle finit donc par dépasser n.

Correction :
On sort de la boucle pour la plus petite valeur de i telle que $i! \geq n$.

Le test $i! = n$ permet de bien savoir si n est une factorielle.

Complexité :

On a 2 affectations en début de programme. Le test est un peu chargé en opérations mais pour chaque itération, le nombre d'opérations élémentaires est identique tant que le test est faux.

Complexité :
On a 2 affectations en début de programme. Le test est un peu chargé en opérations mais pour chaque itération, le nombre d'opérations élémentaires est identique tant que le test est faux.

Le problème est de déterminer le nombre d'opérations nb nécessaires en fonction de n. Le coût est donc un $\Theta(nb)$ mais impossible de connaître nb en fonction de n.