Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?
3. complexité : evolution du temps d'exécution de l'algorithme en fonction de la taille des données (prédiction de temps de calcul, comparaisons des performances d'algorithmes résolvant le même problème.)

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?
3. complexité : evolution du temps d'exécution de l'algorithme en fonction de la taille des données (prédiction de temps de calcul, comparaisons des performances d'algorithmes résolvant le même problème.)

L'algorithme étudié doit avoir une spécification précise (entrées, sorties, préconditions, postconditions, effets de bord). On parle d'algorithmes (et non de programmes) car ces questions sont indépendantes de l'implémentation dans un langage de programmation quelconque.

Définitions

Définitions

- On dit qu'un algorithme termine si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.

Définitions

- On dit qu'un algorithme termine si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :

Définitions

- On dit qu'un algorithme termine si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,

Définitions

- On dit qu'un algorithme termine si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,
  - qui décroît *strictement* à chaque passage dans la boucle.

Définitions

- On dit qu'un algorithme termine si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,
  - qui décroît *strictement* à chaque passage dans la boucle.
- Pour un algorithme récursif, un variant est une quantité :

Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,
  - qui décroît *strictement* à chaque passage dans la boucle.
- Pour un algorithme récursif, un **variant** est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,

Définitions

- On dit qu'un algorithme termine si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,
  - qui décroît *strictement* à chaque passage dans la boucle.
- Pour un algorithme récursif, un variant est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,
  - qui décroît *strictement* à chaque appel récursif.

Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,
  - qui décroît *strictement* à chaque passage dans la boucle.
- Pour un algorithme récursif, un **variant** est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - *strictement* positive,
  - qui décroît *strictement* à chaque appel récursif.

Preuve de la terminaison d'un algorithme

Pour prouver la terminaison d'un algorithme, il suffit de trouver un **variant de boucle** pour chaque boucle non bornée qu'il contient. Et un **variant** pour chaque fonction récursive.

Exemple 1

On considère la fonction ci-dessous :

```
/* Renvoie le quotient dans la division euclidienne de a par b avec
↪  a et b deux entiers naturels et b non nul  */
int quotient(int a,  int b)
{   assert (a>=0 && b>0);
    int q = 0;
    while (a - b >= 0)
    {
        a = a - b;
        q = q + 1;
    }
    return q;
}
```

Exemple 1

On considère la fonction ci-dessous :

```c
/* Renvoie le quotient dans la division euclidienne de a par b avec a et b deux entiers naturels et b non nul  */
int quotient(int a,  int b)
{   assert (a>=0 && b>0);
    int q = 0;
    while (a - b >= 0)
    {
        a = a - b;
        q = q + 1;
    }
    return q;
}
```

En trouvant un variant de boucle, prouver la terminaison de cette fonction.

Correction de l'exemple 1

Montrons que la variable a est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

Correction de l'exemple 1

Montrons que la variable a est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de a est un entier naturel (précondition)

Correction de l'exemple 1

Montrons que la variable a est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de a est un entier naturel (précondition)
- A chaque tour de boucle la valeur de a diminue (de b> 0).

Correction de l'exemple 1

Montrons que la variable `a` est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de `a` est un entier naturel (précondition)
- A chaque tour de boucle la valeur de `a` diminue (de `b`> 0).
- La nouvelle valeur de `a` est `a-b` qui est garantie positive par condition d'entrée dans la boucle

Correction de l'exemple 1

Montrons que la variable a est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de a est un entier naturel (précondition)
- A chaque tour de boucle la valeur de a diminue (de `b`> 0).
- La nouvelle valeur de a est `a-b` qui est garantie positive par condition d'entrée dans la boucle

Les trois éléments ci-dessus prouvent que la variable a est un variant de la boucle `while` de ce programme, par conséquent cette boucle se termine.

Exemple 2

On considère la fonction ci-dessous :

```
let rec est_dans element liste =
  match liste with
  | [] -> false
  | head::tail -> head = element || est_dans element tail
```

Exemple 2

On considère la fonction ci-dessous :

```
let rec est_dans element liste =
  match liste with
  | [] -> false
  | head::tail -> head = element || est_dans element tail
```

Prouver que cette fonction récursive termine

Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque appel récursif.

Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque appel récursif.

- La longueur d'une liste est à valeur dans $\mathbb{N}$

Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque appel récursif.

- La longueur d'une liste est à valeur dans $\mathbb{N}$
- A chaque appel récursif, on enlève un élément de `liste` (sa tête) et donc la taille de la liste diminue de 1.

Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque appel récursif.

- La longueur d'une liste est à valeur dans $\mathbb{N}$
- A chaque appel récursif, on enlève un élément de `liste` (sa tête) et donc la taille de la liste diminue de 1.

Les deux éléments ci-dessus prouvent que la longueur de la liste est un variant et que donc cette fonction récursive termine.

Exemple 3

- Rappeler l'algorithme d'Euclide pour la calcul du PGCD de deux entiers naturels a et$b$.

Exemple 3

- Rappeler l'algorithme d'Euclide pour la calcul du PGCD de deux entiers naturels a et$b$.
- Faire fonctionner cet algorithme pour calculer le PGCD de 72 et 132.

Exemple 3

- Rappeler l'algorithme d'Euclide pour la calcul du PGCD de deux entiers naturels a et$b$.
- Faire fonctionner cet algorithme pour calculer le PGCD de 72 et 132.
- Ecrire une implémentation itérative en C de cet algorithme avec une fonction de signature `int pgcd(int a, int b)`.

Exemple 3

- Rappeler l'algorithme d'Euclide pour la calcul du PGCD de deux entiers naturels a et$b$.
- Faire fonctionner cet algorithme pour calculer le PGCD de 72 et 132.
- Ecrire une implémentation itérative en C de cet algorithme avec une fonction de signature `int pgcd(int a, int b)`.
- Ecrire une implémentation récursive en Ocaml de cet algorithme.

Exemple 3

- Rappeler l'algorithme d'Euclide pour la calcul du PGCD de deux entiers naturels a et$b$.
- Faire fonctionner cet algorithme pour calculer le PGCD de 72 et 132.
- Ecrire une implémentation itérative en C de cet algorithme avec une fonction de signature `int pgcd(int a, int b)`.
- Ecrire une implémentation récursive en Ocaml de cet algorithme.
- Prouver la terminaison dans les deux cas.

Remarque

Dans les exemples ci-dessus la mise en évidence du variant est facile (et ce sera le cas en général dans nos algorithmes). Mais, les preuves de terminaison sont loin d'être toujours aussi évidentes.

Remarque

Dans les exemples ci-dessus la mise en évidence du variant est facile (et ce sera le cas en général dans nos algorithmes). Mais, les preuves de terminaison sont loin d'être toujours aussi évidentes.
Par exemple, si on considère la fonction suivante :

```
let rec syracuse n =
  if n=1 then 1 else
    if n mod 2 = 0 then syracuse (n/2) else syracuse(3*n+1)
```

Remarque

Dans les exemples ci-dessus la mise en évidence du variant est facile (et ce sera le cas en général dans nos algorithmes). Mais, les preuves de terminaison sont loin d'être toujours aussi évidentes.
Par exemple, si on considère la fonction suivante :

```
let rec syracuse n =
  if n=1 then 1 else
    if n mod 2 = 0 then syracuse (n/2) else syracuse(3*n+1)
```

Prouver sa terminaison reviendrait à prouver la conjecture de syracuse qui résiste aux mathématiciens depuis un siècle !

Correction d'un algorithme

On dira qu'un algorithme est correct

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.
On parle de correction partielle quand le résultat est correct lorsque l'algorithme s'arrête. Et de correction totale lorsque la correction est partielle et que l'algorithme se termine.

Tests et correction

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.
On parle de correction partielle quand le résultat est correct lorsque l'algorithme s'arrête. Et de correction totale lorsque la correction est partielle et que l'algorithme se termine.

Tests et correction

Des tests ne permettent pas de prouver qu'un algorithme est correct.

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.
On parle de correction partielle quand le résultat est correct lorsque l'algorithme s'arrête. Et de correction totale lorsque la correction est partielle et que l'algorithme se termine.

Tests et correction

Des tests ne permettent pas de prouver qu'un algorithme est correct. En effet, ils ne permettent de valider le comportement de l'algorithme que dans quelques cas particuliers et jamais dans le cas général

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

La méthode est similaire à une récurrence mathématique (les deux étapes précédentes correspondent à l'initialisation et à l'hérédité).

Exemple 1

On considère la fonction ci-dessous :

```c
/* Renvoie le nombre d'occurrence de elt dans tab */
int nb_occurence(int elt, int tab[], int size)
{
    int nb = 0;
    for (int i=0; i<size; i++)
    {
        if (tab[i]==elt)
        {
            nb = nb + 1;
        }
    }
    return nb;
}
```

Exemple 1

On considère la fonction ci-dessous :

```c
/* Renvoie le nombre d'occurrence de elt dans tab */
int nb_occurence(int elt, int tab[], int size)
{
    int nb = 0;
    for (int i=0; i<size; i++)
    {
        if (tab[i]==elt)
        {
            nb = nb + 1;
        }
    }
    return nb;
}
```

En trouvant un invariant de boucle, montrer qu'à la sortie de la boucle, la variable `cpt` contient le nombre de fois où `elt` apparaît dans `tab`

Correction de l'exemple 1

Correction de l'exemple 1

Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans le sous tableau* `{tab[0], ... tab[i-1]}` » est un invariant de boucle.

Correction de l'exemple 1

Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans le sous tableau* `{tab[0], ... tab[i-1]}` » est un invariant de boucle.

Correction de l'exemple 1

Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans le sous tableau* `{tab[0], ... tab[i-1]}` » est un invariant de boucle.

- En entrant dans la boucle ($i = 0$) la propriété est vraie car le sous tableau est vide et `cpt` vaut 0.

Correction de l'exemple 1

Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans le sous tableau* `{tab[0], ... tab[i-1]}` » est un invariant de boucle.

- En entrant dans la boucle ($i = 0$) la propriété est vraie car le sous tableau est vide et `cpt` vaut 0.
- A chaque tour de boucle, on ajoute 1 au compteur lorsque `elt=tab[i]` donc si cette variable contenait le nombre d'occurrence de `elt` du sous tableau `{tab[0], ... tab[i-1]}` alors elle contiendra le nombre d'occurrence de `elt` du sous tableau `{tab[0], ... tab[i]}`. Donc la propriété reste vraie à chaque tour de boucle.

Correction de l'exemple 1

Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans le sous tableau* `{tab[0], ... tab[i-1]}` » est un invariant de boucle.

- En entrant dans la boucle ($i = 0$) la propriété est vraie car le sous tableau est vide et `cpt` vaut 0.
- A chaque tour de boucle, on ajoute 1 au compteur lorsque `elt=tab[i]` donc si cette variable contenait le nombre d'occurrence de `elt` du sous tableau `{tab[0], ... tab[i-1]}` alors elle contiendra le nombre d'occurrence de `elt` du sous tableau `{tab[0], ... tab[i]}`. Donc la propriété reste vraie à chaque tour de boucle.

Cette propriété est donc bien un invariant de boucle.

Correction de l'exemple 1

Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans le sous tableau* `{tab[0], ... tab[i-1]}` » est un invariant de boucle.

- En entrant dans la boucle ($i = 0$) la propriété est vraie car le sous tableau est vide et `cpt` vaut 0.
- A chaque tour de boucle, on ajoute 1 au compteur lorsque `elt=tab[i]` donc si cette variable contenait le nombre d'occurrence de `elt` du sous tableau `{tab[0], ... tab[i-1]}` alors elle contiendra le nombre d'occurrence de `elt` du sous tableau `{tab[0], ... tab[i]}`. Donc la propriété reste vraie à chaque tour de boucle.

Cette propriété est donc bien un invariant de boucle. L'invariant de boucle reste vraie en sortie de boucle ce qui prouve que l'algorithme est correct.

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir en effectuant une preuve par récurrence.

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir en effectuant une preuve par récurrence.

Exemple 2 : liste des termes pairs

On considère la fonction récursive suivante qui renvoie la liste des termes pairs.

```
let rec pairs liste =
  match liste with
  | [] -> []
  | head :: tail -> let ptail = pairs tail in
    if head mod 2 = 0 then head::ptail else pairs ptail
```

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir en effectuant une preuve par récurrence.

Exemple 2 : liste des termes pairs

On considère la fonction récursive suivante qui renvoie la liste des termes pairs.

```
let rec pairs liste =
  match liste with
  | [] -> []
  | head :: tail -> let ptail = pairs tail in
    if head mod 2 = 0 then head::ptail else pairs ptail
```

On note $P(n)$, la propriété : « *pour une liste de taille n,* `pairs` *renvoie la liste des termes pairs de cette liste* ». Alors :

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir en effectuant une preuve par récurrence.

Exemple 2 : liste des termes pairs

On considère la fonction récursive suivante qui renvoie la liste des termes pairs.

```
let rec pairs liste =
  match liste with
  | [] -> []
  | head :: tail -> let ptail = pairs tail in
    if head mod 2 = 0 then head::ptail else pairs ptail
```

On note $P(n)$, la propriété : « *pour une liste de taille n,* `pairs` *renvoie la liste des termes pairs de cette liste* ». Alors :

- $P(0)$ est vérifiée d'après le cas de base

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir en effectuant une preuve par récurrence.

Exemple 2 : liste des termes pairs

On considère la fonction récursive suivante qui renvoie la liste des termes pairs.

```
let rec pairs liste =
  match liste with
  | [] -> []
  | head :: tail -> let ptail = pairs tail in
    if head mod 2 = 0 then head::ptail else pairs ptail
```

On note $P(n)$, la propriété : « *pour une liste de taille n,* `pairs` *renvoie la liste des termes pairs de cette liste* ». Alors :

- $P(0)$ est vérifiée d'après le cas de base
- On suppose que $P(n)$ vérifié au rang n, et on considère une liste de taille $n+1$ notée `h::t`. Comme `t` est de taille n, on lui applique l'hypothèse de récurrence et `pair t` renvoie bien la liste des termes pairs.

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir en effectuant une preuve par récurrence.

Exemple 2 : liste des termes pairs

On considère la fonction récursive suivante qui renvoie la liste des termes pairs.

```
let rec pairs liste =
  match liste with
  | [] -> []
  | head :: tail -> let ptail = pairs tail in
    if head mod 2 = 0 then head::ptail else pairs ptail
```

On note $P(n)$, la propriété : « *pour une liste de taille n,* `pairs` *renvoie la liste des termes pairs de cette liste* ». Alors :

- $P(0)$ est vérifiée d'après le cas de base
- On suppose que $P(n)$ vérifié au rang n, et on considère une liste de taille $n+1$ notée `h::t`. Comme `t` est de taille n, on lui applique l'hypothèse de récurrence et `pair t` renvoie bien la liste des termes pairs. La formule de récursivité permet alors de conclure que $P(n+1)$ est vérifiée puisque `pair h::t` renvoie h : :(pair t) si `h` est pair et `pair t` sinon.

Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n
Sorties : Booléen indiquant si $a \in t$.

```
pour i ← 0 à n − 1 faire
    si t[i] = a alors
        return Vrai
    fin
fin
return Faux
```

Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n
Sorties : Booléen indiquant si $a \in t$.

```
pour i ← 0 à n − 1 faire
    si t[i] = a alors
        return Vrai
    fin
fin
return Faux
```

1. Combien de comparaisons effectue cet algorithme dans le meilleur des cas ?

Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n

Sorties : Booléen indiquant si $a \in t$.

```
pour i ← 0 à n − 1 faire
    si t[i] = a alors
        return Vrai
    fin
fin
return Faux
```

1. Combien de comparaisons effectue cet algorithme dans le meilleur des cas ?
2. Même question dans le pire des cas.

Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n

Sorties : Booléen indiquant si $a \in t$.

```
pour i ← 0 à n − 1 faire
    si t[i] = a alors
        return Vrai
    fin
fin
return Faux
```

1. Combien de comparaisons effectue cet algorithme dans le meilleur des cas ?
2. Même question dans le pire des cas.
3. Que dire du cas où on recherche un élément a présent en un seul exemplaire dans le tableau t en supposant les positions équiprobables ?

Correction

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X = k) = \dfrac{1}{n}$.
   Donc,

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X=k)=\dfrac{1}{n}$.
   Donc,

$$E(X) = \sum_{k=1}^{n} k\,\frac{1}{n}$$

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X=k)=\dfrac{1}{n}$. Donc,

$$\begin{aligned} E(X) &= \sum_{k=1}^{n} k\,\frac{1}{n} \\ E(X) &= \frac{n+1}{2} \end{aligned}$$

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X = k) = \dfrac{1}{n}$.
   Donc,

$$\begin{aligned} E(X) &= \sum_{k=1}^{n} k\,\frac{1}{n} \\ E(X) &= \frac{n+1}{2} \end{aligned}$$

Le nombre de comparaisons varie donc avec les données du problème.

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X=k)=\dfrac{1}{n}$.
   Donc,
   $$\begin{aligned} E(X) &= \sum_{k=1}^{n} k\,\frac{1}{n} \\ E(X) &= \frac{n+1}{2} \end{aligned}$$

Le nombre de comparaisons varie donc avec les données du problème.
On peut cependant toujours *majorer* le nombre de comparaisons, qui reste inférieur dans tous les cas à Kn où K est une constante et n la taille du tableau.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.
  Dans l'exemple bien que l'algorithme puisse parfois répondre avec une seule comparaison c'est le pire des cas qui nous intéresse.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.
  Dans l'exemple bien que l'algorithme puisse parfois répondre avec une seule comparaison c'est le pire des cas qui nous intéresse.
- on fournit une majoration asymptotique c'est à dire à une constante multiplicative près et à partir d'un certain rang.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.
  Dans l'exemple bien que l'algorithme puisse parfois répondre avec une seule comparaison c'est le pire des cas qui nous intéresse.
- on fournit une majoration asymptotique c'est à dire à une constante multiplicative près et à partir d'un certain rang.
  Dans l'exemple précédent la majoration était Kn.

Définition

La complexité d'un algorithme est une mesure de son efficacité.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

Ces deux éléments varient en fonction de la taille et de la nature des données, on donne donc généralement une majoration dans le pire des cas.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

Ces deux éléments varient en fonction de la taille et de la nature des données, on donne donc généralement une majoration dans le pire des cas.

Remarque

On peut aussi parler de la complexité en moyenne, qui s'intéresse au nombre moyen d'operations effectuées par un algorithme sur un ensemble d'entrées de taille n.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations . . .

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations . . .
- On exprime le coût de l'algorithme pour un entrée de taille `n` en nombre d'opérations élémentaires nécessaires à sa réalisation.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations ...
- On exprime le coût de l'algorithme pour un entrée de taille `n` en nombre d'opérations élémentaires nécessaires à sa réalisation.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations ...
- On exprime le coût de l'algorithme pour un entrée de taille `n` en nombre d'opérations élémentaires nécessaires à sa réalisation.
  - La fonction `let f x = x*x + 2*x + 3;;` demande 5 opérations quelque soit la taille de l'entrée `x`.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations ...
- On exprime le coût de l'algorithme pour un entrée de taille `n` en nombre d'opérations élémentaires nécessaires à sa réalisation.
  - La fonction `let f x = x**x + 2*x + 3;;` demande 5 opérations quelque soit la taille de l'entrée `x`.
  - La fonction suivante en C :

```c
int somme(int tab[], int size){
    int s = 0;
    for (int i=0;i<size;i++){
        s = s + tab[i];}
    return s;}
```

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations . . .
- On exprime le coût de l'algorithme pour un entrée de taille `n` en nombre d'opérations élémentaires nécessaires à sa réalisation.
  - La fonction `let f x = x*x + 2*x + 3;;` demande 5 opérations quelque soit la taille de l'entrée `x`.
  - La fonction suivante en C :

```c
int somme(int tab[], int size){
    int s = 0;
    for (int i=0;i<size;i++){
        s = s + tab[i];}
    return s;}
```

demande $4n+3$ opérations où n est la taille du tableau.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.
  On note alors $u = \mathcal{O}(v)$ (ou encore $u \in \mathcal{O}(v)$) et on dit que u est un grand $\mathcal{O}$ de v.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.
  On note alors $u = \mathcal{O}(v)$ (ou encore $u \in \mathcal{O}(v)$) et on dit que u est un grand $\mathcal{O}$ de v.
  « u_n est inférieur à v_n à une constante multiplicative près et pour n assez grand ».

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.
  On note alors $u = \mathcal{O}(v)$ (ou encore $u \in \mathcal{O}(v)$) et on dit que u est un grand $\mathcal{O}$ de v.
  « u_n est inférieur à v_n à une constante multiplicative près et pour n assez grand ».
  Dans l'exemple précédent, $C(n) = 3n + 15$ est un $\mathcal{O}(n)$.

Exemples

- Montrer que (a_n) de terme général $10n + 3$ est un $\mathcal{O}(n)$

Exemples

- Montrer que (a_n) de terme général $10n + 3$ est un $\mathcal{O}(n)$
- Montrer que (b_n) de terme général $n^2 + n + 1$ est un $\mathcal{O}(n^2)$

Exemples

- Montrer que (a_n) de terme général $10n + 3$ est un $\mathcal{O}(n)$
- Montrer que (b_n) de terme général $n^2 + n + 1$ est un $\mathcal{O}(n^2)$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n + \ln(n)$

Exemples

- Montrer que (a_n) de terme général $10n+3$ est un $\mathcal{O}(n)$
  $10n+3 < 11n$ pour $n > 3$
- Montrer que (b_n) de terme général n^2+n+1 est un $\mathcal{O}(n^2)$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n+\ln(n)$

Exemples

- Montrer que (a_n) de terme général $10n + 3$ est un $\mathcal{O}(n)$
  $10n + 3 < 11n$ pour $n > 3$
- Montrer que (b_n) de terme général $n^2 + n + 1$ est un $\mathcal{O}(n^2)$
  $n^2 + n + 1 < 2n^2$ pour $n > 2$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n + \ln(n)$

Exemples

- Montrer que (a_n) de terme général $10n+3$ est un $\mathcal{O}(n)$
  $10n+3 < 11n$ pour $n > 3$
- Montrer que (b_n) de terme général n^2+n+1 est un $\mathcal{O}(n^2)$
  $n^2+n+1 < 2n^2$ pour $n > 2$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n+\ln(n)$
  Comme $\ln(n) < n$, $c_n < 8n$ et donc (c_n) est un $\mathcal{O}(n)$.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».
Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).
On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».
Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).
On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

- $u_n = \Omega(v_n)$ s'il existe $K \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0, u_n \geqslant K v_n$, c'est à dire que « (u_n) est au moins de l'ordre de (v_n) »

.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».
Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).
On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

- $u_n = \Omega(v_n)$ s'il existe $K \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0, u_n \geqslant K v_n$, c'est à dire que « (u_n) est au moins de l'ordre de (v_n) »
- $v_n = \Theta(v_n)$ si $u_n = \mathcal{O}(v_n)$ et $v_n = \mathcal{O}(u_n)$, c'est à dire que « (u_n) est de l'ordre de (v_n) »

.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».
Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).
On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

- $u_n = \Omega(v_n)$ s'il existe $K \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0, u_n \geqslant K v_n$, c'est à dire que « (u_n) est au moins de l'ordre de (v_n) »
- $v_n = \Theta(v_n)$ si $u_n = \mathcal{O}(v_n)$ et $v_n = \mathcal{O}(u_n)$, c'est à dire que « (u_n) est de l'ordre de (v_n) »

On utilisera principalement la notation $\mathcal{O}$, en gardant à l'esprit qu'on essaye toujours d'avoir la meilleure majoration.

Complexités usuelles

Complexité	Nom	Exemple

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste
$\mathcal{O}(n\log(n))$	Linéaritmique	Tri fusion

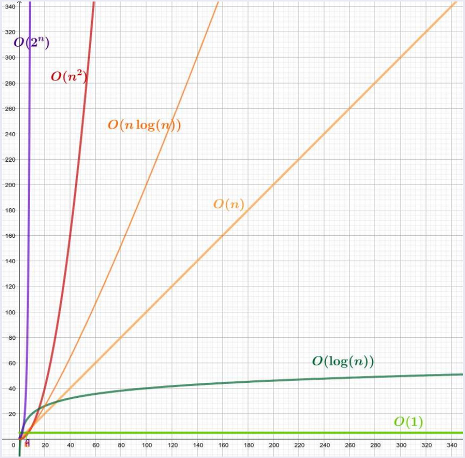
Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste
$\mathcal{O}(n\log(n))$	Linéaritmique	Tri fusion
$\mathcal{O}(n^2)$	Quadratique	Tri par insertion d'une liste

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste
$\mathcal{O}(n\log(n))$	Linéaritmique	Tri fusion
$\mathcal{O}(n^2)$	Quadratique	Tri par insertion d'une liste
$\mathcal{O}(2^n)$	Exponentielle	Algorithme par force brute pour le sac à dos

Représentation graphique



Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✖ un temps de calcul inaccessible :

	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$					
$\mathcal{O}(n\log(n))$					
$\mathcal{O}(n^2)$					
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq$ 10s
$\mathcal{O}(n\log(n))$					
$\mathcal{O}(n^2)$					
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq 10$s
$\mathcal{O}(n\log(n))$	✔	✔	✔	✔	$\simeq 1,5$ mn
$\mathcal{O}(n^2)$					
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n=10$	$n=100$	$n=1000$	$n=10^6$	$n=10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq 10$s
$\mathcal{O}(n\log(n))$	✔	✔	✔	✔	$\simeq 1,5$ mn
$\mathcal{O}(n^2)$	✔	✔	✔	$\simeq 3$ h	$\simeq 300$ ans
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n=10$	$n=100$	$n=1000$	$n=10^6$	$n=10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq 10$s
$\mathcal{O}(n\log(n))$	✔	✔	✔	✔	$\simeq 1,5$ mn
$\mathcal{O}(n^2)$	✔	✔	✔	$\simeq 3$ h	$\simeq 300$ ans
$\mathcal{O}(2^n)$	✔	✘	✘	✘	✘

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.
  La taille des données a été multiplié par 250, la complexité étant quadratique le temps de calcul sera approximativement multiplié par $250^2 = 62500$

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.
  La taille des données a été multiplié par 250, la complexité étant quadratique le temps de calcul sera approximativement multiplié par $250^2 = 62500$
  $0.07 \times 62\,500 = 4375$, on peut donc prévoir un temps de calcul d'environ $4\,375$ secondes, c'est-à-dire près d'une heure et 15 minutes !

Equation de complexité

Dans le cas des fonction récursives, la complexité pour une entrée de taille n s'exprime à partir de complexité pour des tailles inférieures. On est donc amené à résoudre une *équation de complexité*.

Equation de complexité

Dans le cas des fonction récursives, la complexité pour une entrée de taille n s'exprime à partir de complexité pour des tailles inférieures. On est donc amené à résoudre une *équation de complexité*.

Exemple

Par exemple si on considère la version récursive du calcul de la somme d'une liste d'entiers en OCaml :

```
let rec somme l =
  match l with
  | [] -> 0
  | h::t -> h + somme t
```

Alors on a $C(n) = C(n-1) + a$, et donc $C(n)$ est arithmétique de raison a et $C(n)$ est un $O(n)$.

Les tours de Hanoï

On rappelle que le jeu des tours de Hanoï peut être résolu de façon élégante par récursion. On note $T(n)$ le nombre de mouvement minimal nécessaire afin de résoudre Hanoï avec n disques en utilisant l'algorithme récursif.

1. Déterminer $T(1)$
2. Exprimer $T(n)$ en fonction $T(n-1)$
3. En déduire la complexité de l'algorithme.

Enoncé : recherche dichotomique

On reprend l'exemple de la recherche dichotomique dans un tableau trié :

```
bool recherche_dichotomique(int elt, int tab[], int size){
    int deb = 0;
    int fin = size - 1;
    int milieu;
    while (fin - deb >= 0){
        milieu = (deb + fin) / 2;
        if (tab[milieu] == elt){
            return true;}
        else{
            if (tab[milieu] < elt){
                deb = milieu + 1;}
            else{
            fin = milieu - 1;}}
    }
    return false;}
```

Enoncé : recherche dichotomique

On reprend l'exemple de la recherche dichotomique dans un tableau trié :

```c
bool recherche_dichotomique(int elt, int tab[], int size){
    int deb = 0;
    int fin = size - 1;
    int milieu;
    while (fin - deb >= 0){
        milieu = (deb + fin) / 2;
        if (tab[milieu] == elt){
            return true;}
        else{
            if (tab[milieu] < elt){
                deb = milieu + 1;}
            else{
            fin = milieu - 1;}}
    }
    return false;}
```

- Prouver que cet algorithme termine

Enoncé : recherche dichotomique

On reprend l'exemple de la recherche dichotomique dans un tableau trié :

```c
bool recherche_dichotomique(int elt, int tab[], int size){
    int deb = 0;
    int fin = size - 1;
    int milieu;
    while (fin - deb >= 0){
        milieu = (deb + fin) / 2;
        if (tab[milieu] == elt){
            return true;}
        else{
            if (tab[milieu] < elt){
                deb = milieu + 1;}
            else{
            fin = milieu - 1;}}
    }
    return false;}
```

- Prouver que cet algorithme termine
- Prouver qu'il est correct

Enoncé : recherche dichotomique

On reprend l'exemple de la recherche dichotomique dans un tableau trié :

```
bool recherche_dichotomique(int elt, int tab[], int size){
    int deb = 0;
    int fin = size - 1;
    int milieu;
    while (fin - deb >= 0){
        milieu = (deb + fin) / 2;
        if (tab[milieu] == elt){
            return true;}
        else{
            if (tab[milieu] < elt){
                deb = milieu + 1;}
            else{
            fin = milieu - 1;}}
    }
    return false;}
```

- Prouver que cet algorithme termine
- Prouver qu'il est correct
- Donner sa complexité.

Correction : recherche dichotomique

Correction : recherche dichotomique

- Montrons que les valeurs prises par `fin - deb` est un variant de boucle.
  - `fin-deb` est positif à l'entrée dans la boucle (le tableau est supposé non vide)
  - `fin-deb` décroît strictement à chaque itération car soit `fin` diminue, soit deb augmente

Correction : recherche dichotomique

- Montrons que les valeurs prises par `fin - deb` est un variant de boucle.
  - `fin-deb` est positif à l'entrée dans la boucle (le tableau est supposé non vide)
  - `fin-deb` décroît strictement à chaque itération car soit `fin` diminue, soit deb augmente
- Si la fonction renvoie `true` alors l'élément cherché est bien présent dans le tableau car le test `tab[milieu]==elt` est vraie. Montrons maintenant que si la fonction renvoie `false` alors l'élément n'est pas dans le tableau. Pour cela on montre la propriété suivante : P : « Si `elt` est dans `tab` alors il se trouve entre les indices `deb` et `fin` ». En effet, :

Correction : recherche dichotomique

- Montrons que les valeurs prises par `fin - deb` est un variant de boucle.
  - `fin-deb` est positif à l'entrée dans la boucle (le tableau est supposé non vide)
  - `fin-deb` décroît strictement à chaque itération car soit `fin` diminue, soit deb augmente
- Si la fonction renvoie `true` alors l'élément cherché est bien présent dans le tableau car le test `tab[milieu]==elt` est vraie. Montrons maintenant que si la fonction renvoie `false` alors l'élément n'est pas dans le tableau. Pour cela on montre la propriété suivante : P : « Si `elt` est dans `tab` alors il se trouve entre les indices `deb` et `fin` ». En effet, :
  - Cette propriété est vraie à l'entrée dans la boucle puisque `deb=0` et `fin=size-1` la totalité du tableau est couverte.

Correction : recherche dichotomique

- Montrons que les valeurs prises par `fin - deb` est un variant de boucle.
  - `fin-deb` est positif à l'entrée dans la boucle (le tableau est supposé non vide)
  - `fin-deb` décroît strictement à chaque itération car soit `fin` diminue, soit deb augmente
- Si la fonction renvoie `true` alors l'élément cherché est bien présent dans le tableau car le test `tab[milieu]==elt` est vraie. Montrons maintenant que si la fonction renvoie `false` alors l'élément n'est pas dans le tableau. Pour cela on montre la propriété suivante : P : « Si `elt` est dans `tab` alors il se trouve entre les indices `deb` et `fin` ». En effet, :
  - Cette propriété est vraie à l'entrée dans la boucle puisque `deb=0` et `fin=size-1` la totalité du tableau est couverte.
  - Cette propriété reste vraie à chaque tour de boucle car puisque le tableau est trié, si `elt` est strictement plus grand que `tab[milieu]` alors il se situe forcément après l'indice `milieu` (et strictement avant dans le cas contraire)

Correction : recherche dichotomique

1. Implémentation itérative :

Correction : recherche dichotomique

1. Implémentation itérative :
    - La boucle ne contient que des opérations élémentaires, le coût d'un tour de boucle est donc $O(1)$.

Correction : recherche dichotomique

1. Implémentation itérative :
   - La boucle ne contient que des opérations élémentaires, le coût d'un tour de boucle est donc $O(1)$.
   - La taille de l'intervalle de recherche est initialement la taille `n` du tableau et est divisé par deux à chaque itération. Donc le nombre de tours de boucle est un $O(\log(n))$.

Correction : recherche dichotomique

1. Implémentation itérative :
   - La boucle ne contient que des opérations élémentaires, le coût d'un tour de boucle est donc $O(1)$.
   - La taille de l'intervalle de recherche est initialement la taille `n` du tableau et est divisé par deux à chaque itération. Donc le nombre de tours de boucle est un $O(\log(n))$.
   - En conclusion, la recherche dichotomique a une complexité logarithmique.

Correction : recherche dichotomique

1. Implémentation itérative :
   - La boucle ne contient que des opérations élémentaires, le coût d'un tour de boucle est donc $O(1)$.
   - La taille de l'intervalle de recherche est initialement la taille `n` du tableau et est divisé par deux à chaque itération. Donc le nombre de tours de boucle est un $O(\log(n))$.
   - En conclusion, la recherche dichotomique a une complexité logarithmique.
2. Implémentation récursive :
   Dans ce cas, on obtient l'équation de complexité $C(n) = C(n/2) + 1$ qui conduit au même résultat.

Tri sélection

- Rappeler l'algorithme du tri par sélection

Tri sélection

- Rappeler l'algorithme du tri par sélection
  On note n la taille du tableau, pour chaque entier $i = 0 \dots n-1$, on échange l'élément situé à l'indice i avec le minimum du tableau depuis l'indice i.

Tri sélection

- Rappeler l'algorithme du tri par sélection
  On note n la taille du tableau, pour chaque entier $i = 0 \dots n-1$, on échange l'élément situé à l'indice i avec le minimum du tableau depuis l'indice i.
- Montrer que l'algorithme termine

Tri sélection

- Rappeler l'algorithme du tri par sélection
  On note n la taille du tableau, pour chaque entier $i = 0 \dots n-1$, on échange l'élément situé à l'indice i avec le minimum du tableau depuis l'indice i.
- Montrer que l'algorithme termine
  L'algorithme ne contient pas de boucle non bornées donc sa terminaison est garantie.

Tri sélection

- Rappeler l'algorithme du tri par sélection
  On note n la taille du tableau, pour chaque entier $i = 0 \ldots n-1$, on échange l'élément situé à l'indice i avec le minimum du tableau depuis l'indice i.
- Montrer que l'algorithme termine
  L'algorithme ne contient pas de boucle non bornées donc sa terminaison est garantie.
- Montrer qu'il est correct

Tri sélection

- Rappeler l'algorithme du tri par sélection
  On note n la taille du tableau, pour chaque entier $i = 0 \ldots n - 1$, on échange l'élément situé à l'indice i avec le minimum du tableau depuis l'indice i.
- Montrer que l'algorithme termine
  L'algorithme ne contient pas de boucle non bornées donc sa terminaison est garantie.
- Montrer qu'il est correct
- Déterminer sa complexité

Correction du tri sélection

Pour montrer que l'algorithme est correct, on prouve l'invariant : I : « Les i premiers éléments du tableau sont ceux du tableau trié ».

Correction du tri sélection

Pour montrer que l'algorithme est correct, on prouve l'invariant : I : « Les i premiers éléments du tableau sont ceux du tableau trié ».

- Pour $i = 0$, la propriété est vraie (aucun élément n'est encore trié)

Correction du tri sélection

Pour montrer que l'algorithme est correct, on prouve l'invariant : I : « Les i premiers éléments du tableau sont ceux du tableau trié ».

- Pour $i = 0$, la propriété est vraie (aucun élément n'est encore trié)
- En supposant l'invariant vraie au début d'un tour de boucle (c'est-à-dire les i premiers éléments du tableau sont ceux du tableau trié), on montre qu'il est conservé. L'algorithme consiste à placer le minimum des éléments restants à l'indice $i+1$, cet élément est bien celui d'indice $i+1$ dans le tableau trié (il est supérieur aux éléments situés avant et inférieur à ceux qui restent à trier). Et donc l'invariant est conservé.

Complexité du tri sélection

En notant n, la taille du tableau

- Chaque recherche de minimum parcourt au plus la totalité du tableau et est donc un $O(n)$.

Complexité du tri sélection

En notant n, la taille du tableau

- Chaque recherche de minimum parcourt au plus la totalité du tableau et est donc un $O(n)$.
- Cette recherche est effectuée $n-1$ fois et est donc aussi un $O(n)$.

Complexité du tri sélection

En notant n, la taille du tableau

- Chaque recherche de minimum parcourt au plus la totalité du tableau et est donc un $O(n)$.
- Cette recherche est effectuée $n-1$ fois et est donc aussi un $O(n)$.
- En conclusion la tri par sélection a une complexité en $O(n^2)$.

Tri fusion

- Rappeler le principe de l'algorithme.

Tri fusion

- Rappeler le principe de l'algorithme.
- Prouver la terminaison de l'algorithme.

Tri fusion

- Rappeler le principe de l'algorithme.
- Prouver la terminaison de l'algorithme.
- Montrer qu'il est correct.

Tri fusion

- Rappeler le principe de l'algorithme.
- Prouver la terminaison de l'algorithme.
- Montrer qu'il est correct.
- Déterminer sa complexité.

Définition

On considère une structure de données munie d'un ensemble d'opérations $\mathcal{P}$. Sur une instance de cette structure de données on effectue une suite d'opérations $\{p_1, \dots, p_n\}$ où $p_i \in \mathcal{P}$ pour $i \in [\![1; n]\!]$. En notant c_i le coût en nombre d'operations élémentaires de p_i,la complexité amortie de chacune des opérations p_i est $\dfrac{1}{n}\displaystyle\sum_{k=1}^{n} c_i$.