Algorithmique

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

Algorithmique

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

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

Algorithmique

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

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes 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 suivantes :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes 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. En particulier, le temps d'exécution d'un algorithme sur une entrée donnée sera-t-il « raisonnable » ?

C7 Terminaison, correction, compléxité.

1. ??

Algorithmique

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

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes 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. En particulier, le temps d'exécution d'un algorithme sur une entrée donnée sera-t-il « raisonnable » ?

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.

Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

- A obtenir une preuve mathématique que ces algorithmes se terminent et donc n'entrent jamais dans une boucle infinie quelques soient les données.

Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

- A obtenir une preuve mathématique que ces algorithmes se terminent et donc n'entrent jamais dans une boucle infinie quelques soient les données.
- A obtenir une preuve mathématique que ces algorithmes trient effectivement les listes de nombres données en argument et ce quelques soient leur taille et les valeurs qu'elles contiennent.

Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

- A obtenir une preuve mathématique que ces algorithmes se terminent et donc n'entrent jamais dans une boucle infinie quelques soient les données.
- A obtenir une preuve mathématique que ces algorithmes trient effectivement les listes de nombres données en argument et ce quelques soient leur taille et les valeurs qu'elles contiennent.
- A comparer ces algorithmes en quantifiant leur efficacité (qui peut être mesuré de diverses façons).

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 $\mathbb{N}$.
  - qui décroît strictement à chaque passage dans la boucle.
- Pour un algorithme récursif, un variant est une quantité :
  - à valeurs dans $\mathbb{N}$.
  - 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 :

```
1  /* Renvoie le quotient dans la division euclidienne de a par b avec
   ↪  a et b deux entiers naturels et b non nul  */
2  int quotient(int a,  int b)
3  {   assert (a>=0 && b>0);
4      int q = 0;
5      while (a - b >= 0)
6      {
7          a = a - b;
8          q = q + 1;
9      }
10     return q;
11 }
```

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 :

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

Exemple 2

On considère la fonction ci-dessous :

```
1  let rec est_dans element liste =
2    match liste with
3    | [] -> false
4    | 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.

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 :

```
1  let rec syracuse n =
2    if n=1 then 1 else
3      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 :

```
1  let rec syracuse n =
2    if n=1 then 1 else
3      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

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `tab` et k le nombre de tours de boucle

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `tab` et k le nombre de tours de boucle
Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `tab` » est un invariant de boucle.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `tab` et k le nombre de tours de boucle
Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `tab` » est un invariant de boucle.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `tab` et k le nombre de tours de boucle
Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `tab` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `tab` et k le nombre de tours de boucle
Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `tab` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `tab` et k le nombre de tours de boucle
Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `tab` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Cette propriété est donc bien un invariant de boucle.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `tab` et k le nombre de tours de boucle
Montrons que la propriété :
« `nb` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `tab` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

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 :

- Directement à partir des identités mathématiques issus du code de la fonction.

Exemple 2 : factoriel récursif

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir :

- Directement à partir des identités mathématiques issus du code de la fonction.
- Par un raisonnement inductif, c'est à dire similaire à une récurrence.

Exemple 2 : factoriel récursif

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir :

- Directement à partir des identités mathématiques issus du code de la fonction.
- Par un raisonnement inductif, c'est à dire similaire à une récurrence.

Exemple 2 : factoriel récursif

```
1  let rec fact n =
2    if n = 0 then 1 else n *  fact (n-1)
```

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir :

- Directement à partir des identités mathématiques issus du code de la fonction.
- Par un raisonnement inductif, c'est à dire similaire à une récurrence.

Exemple 2 : factoriel récursif

```
1  let rec fact n =
2    if n = 0 then 1 else n *  fact (n-1)
```

Les identités `fact 0 = 1` et `fact n = n * fact (n-1)` si $n > 0$, correspondent bien à la définition mathématique de la factorielle c'est à dire $0! = 1$ et pour tout $n \in \mathbb{N}^*$, $n! = n \times (n-1)!$, donc cette fonction est correcte

Exemple 2 : liste des termes pairs

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

Exemple 2 : liste des termes pairs

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

```
1  let rec pairs liste =
2    match liste with
3    | [] -> []
4    | head :: tail -> let ptail = pairs tail in
5      if head mod 2 = 0 then head::ptail else pairs ptail
```

Exemple 2 : liste des termes pairs

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

```
1  let rec pairs liste =
2    match liste with
3    | [] -> []
4    | head :: tail -> let ptail = pairs tail in
5      if head mod 2 = 0 then head::ptail else pairs ptail
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant :

Exemple 2 : liste des termes pairs

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

```
1  let rec pairs liste =
2    match liste with
3    | [] -> []
4    | head :: tail -> let ptail = pairs tail in
5      if head mod 2 = 0 then head::ptail else pairs ptail
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : On note $P(n)$, la propriété : « *pour une liste de taille* n, `pairs` *renvoie la liste des termes pairs de cette liste* ». Alors :

Exemple 2 : liste des termes pairs

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

```
1  let rec pairs liste =
2    match liste with
3    | [] -> []
4    | head :: tail -> let ptail = pairs tail in
5      if head mod 2 = 0 then head::ptail else pairs ptail
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : 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

Exemple 2 : liste des termes pairs

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

```
1  let rec pairs liste =
2    match liste with
3    | [] -> []
4    | head :: tail -> let ptail = pairs tail in
5      if head mod 2 = 0 then head::ptail else pairs ptail
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : 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.

Exemple 2 : liste des termes pairs

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

```
1  let rec pairs liste =
2    match liste with
3    | [] -> []
4    | head :: tail -> let ptail = pairs tail in
5      if head mod 2 = 0 then head::ptail else pairs ptail
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : 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.

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 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 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 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.

Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'elt est ou non dans tab

```
1  bool est_dans(int elt, int tab[], int size){
2      for (int i=0; i<size; i=i+1)
3          {if (tab[i]==elt)
4              {return true;}}
5      return false;}
```

Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'elt est ou non dans tab

```
1  bool est_dans(int elt, int tab[], int size){
2      for (int i=0; i<size; i=i+1)
3          {if (tab[i]==elt)
4              {return true;}}
5      return false;}
```

Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'`elt` est ou non dans `tab`

```
1  bool est_dans(int elt, int tab[], int size){
2      for (int i=0; i<size; i=i+1)
3          {if (tab[i]==elt)
4              {return true;}}
5      return false;}
```

1. En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant une seule comparaison.

Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'elt est ou non dans tab

```
1  bool est_dans(int elt, int tab[], int size){
2      for (int i=0; i<size; i=i+1)
3          {if (tab[i]==elt)
4              {return true;}}
5      return false;}
```

1. En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant une seule comparaison.
2. En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant n comparaison.

Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'elt est ou non dans tab

```
1  bool est_dans(int elt, int tab[], int size){
2      for (int i=0; i<size; i=i+1)
3          {if (tab[i]==elt)
4              {return true;}}
5      return false;}
```

1. En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant une seule comparaison.
2. En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant n comparaison.
3. On suppose à présent qu'on cherche un élément a qui se trouve à un seul exemplaire dans le tableau et que les positions sont équiprobables. C'est à dire que pour tout $i \in [\![0; n-1]\!]$ a se trouve à l'indice i avec la probabilité $\frac{1}{n}$. Quel sera le nombre *moyen* de comparaison à effectuer avec de renvoyer le résultat ?

Exemple : recherche simple dans un tableau

Exemple : recherche simple dans un tableau

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

Exemple : recherche simple dans un tableau

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.

Exemple : recherche simple dans un tableau

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 $p(X = k) = \frac{1}{n}$. Donc,

Exemple : recherche simple dans un tableau

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 $p(X = k) = \frac{1}{n}$. Donc,

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

Exemple : recherche simple dans un tableau

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 $p(X = k) = \frac{1}{n}$. Donc,

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

Exemple : recherche simple dans un tableau

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 $p(X = k) = \frac{1}{n}$. Donc,

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

$$E(X) = \frac{n+1}{2}$$

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

Types de complexité

On appelle :

Types de complexité

On appelle :

- complexité dans le meilleur cas, le nombre minimal d'operations effectuées par un algorithme sur une entrée de taille n.

Types de complexité

On appelle :

- complexité dans le meilleur cas, le nombre minimal d'operations effectuées par un algorithme sur une entrée de taille n.
- complexité dans le pire cas, le nombre maximal d'operations effectuées par un algorithme sur une entrée de taille n.

Types de complexité

On appelle :

- complexité dans le meilleur cas, le nombre minimal d'operations effectuées par un algorithme sur une entrée de taille n.
- complexité dans le pire cas, le nombre maximal d'operations effectuées par un algorithme sur une entrée de taille n.
- complexité en moyenne, le nombre moyen d'operations effectuées par un algorithme sur un ensemble d'entrées de taille n.