Complexité des algorithmes : Grand Oh [1]

Salem BENFERHAT

Centre de Recherche en Informatique de Lens (CRIL-CNRS)
email : benferhat@cril.fr

[1] Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Grand Oh

Grand *Oh*

Remarques

- Le nombre d'instructions associé à un algorithme A donne une idée relativement précise du temps d'exécution que peut prendre cet algorithme.

Grand *Oh*

Remarques

- Le nombre d'instructions associé à un algorithme A donne une idée relativement précise du temps d'exécution que peut prendre cet algorithme.
- Lorsque la taille des données est petite, en général il est difficile de voir la différence du temps d'exécution entre les différents algorithmes d'un problème (sauf si le problème est super exponnentiel).

Grand *Oh*

Remarques

- Le nombre d'instructions associé à un algorithme A donne une idée relativement précise du temps d'exécution que peut prendre cet algorithme.
- Lorsque la taille des données est petite, en général il est difficile de voir la différence du temps d'exécution entre les différents algorithmes d'un problème (sauf si le problème est super exponnentiel).
- Par exemple, si la taille d'un tableau à trier est inférieur à 100 éléments, alors il est difficile de voir la différence entre le temps de l'exécution des différents algorithmes de tri.

Grand *Oh*

- on s'intéresse au comportement des algorithmes lorsque la taille des données est grande (on parle d'une analyse asymptotique des algorithmes).

Grand *Oh*

- on s'intéresse au comportement des algorithmes lorsque la taille des données est grande (on parle d'une analyse asymptotique des algorithmes).
- L'idée ensuite est de créer des classes de complexités équivalentes. Intuitivement, deux algorithmes A et B appartiennent à la même classe de complexité, si $\frac{T_A(n)}{T_B(n)}$ tend vers une constante lorsque n est grand. Pas abus de langage, on dit que $T_A(n)$ est de l'ordre de $T_B(n)$ et inversement.

Grand *Oh*

- En particulier, on essaie de se ”débarrasser” de certains détails dans les expressions de $T(n)$. A titre de comparaison, si on demande l'âge d'un adulte : 60 ans et 60ans3secondes sont considérées comme deux réponses équivalentes.

Grand *Oh*

- En particulier, on essaie de se ”débarrasser” de certains détails dans les expressions de $T(n)$. A titre de comparaison, si on demande l'âge d'un adulte : 60 ans et 60ans3secondes sont considérées comme deux réponses équivalentes.
- On identifiera alors un ensemble de classes de complexité remarquables, et on projettera $T(n)$ dans chacune de ces classes.

Grand *Oh*

Exemple

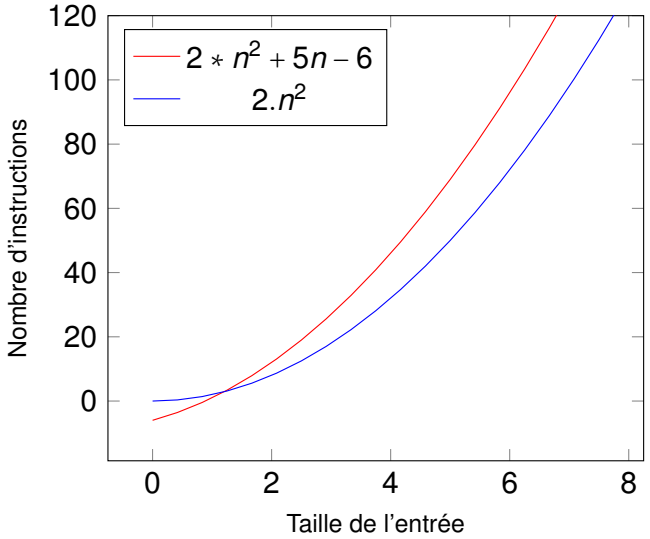
Reprenons notre fonction Tri_sélectif. Nous avons vu ensemble que le nombre d'instructions nécessaire pour trier un tableau de taille n est :

$$T_{\text{Tri\_sél}}(n) = 2n^2 + 5n - 6$$

Comparons $T_{\text{Tri\_sél}}(n)$ avec un certain nombre de fonctions.

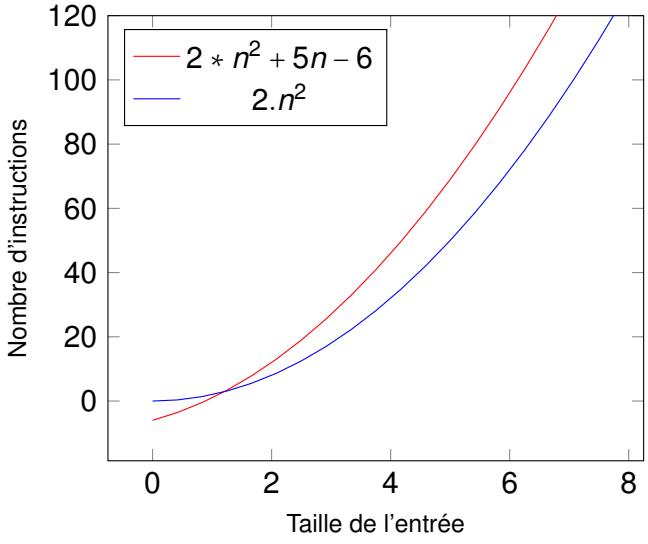
Grand *Oh*

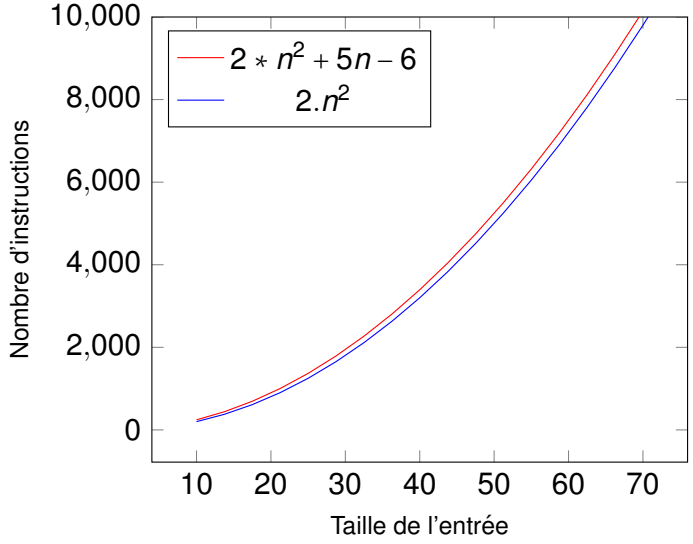
Les figures suivantes donnent la relation entre $T_{\text{Tri\_sél}}(n)$ et $2.n^2$.



Grand *Oh*

Les figures suivantes donnent la relation entre $T_{\text{Tri\_sél}}(n)$ et $2.n^2$.

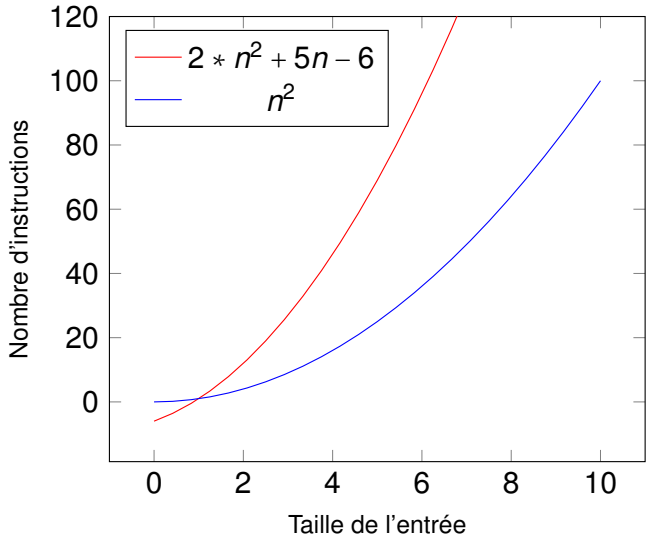




- Lorsque n est grand, alors la différence entre $T_{\text{Tri\_sél}}(n)$ et $2.n^2$ est négligeable (non perceptible à l'exécution).
- C'est-à-dire que si on exécute l'algorithme du tri par sélection et un autre algorithme, qui a comme nombre d'instruction $2.n^2$, alors ils auront le même temps d'exécution (dans les mêmes conditions logistiques et à partir d'un certain n).
- On dira alors que $T_{\text{Tri\_sél}}(n)$ et $2.n^2$ ont le même ordre de grandeur.

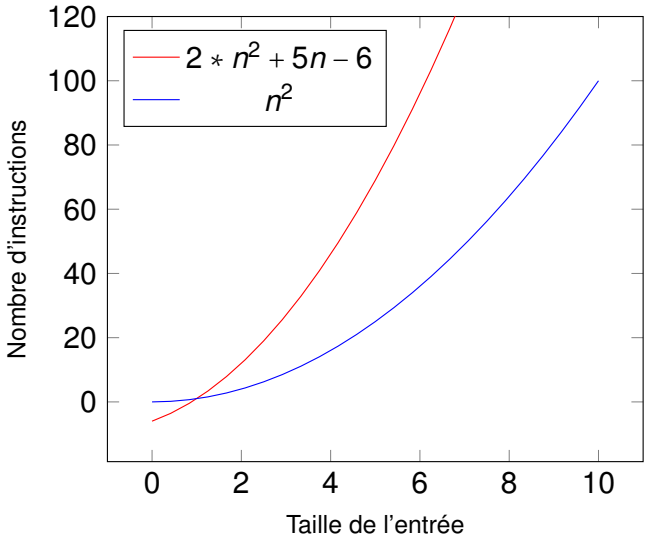
Grand *Oh*

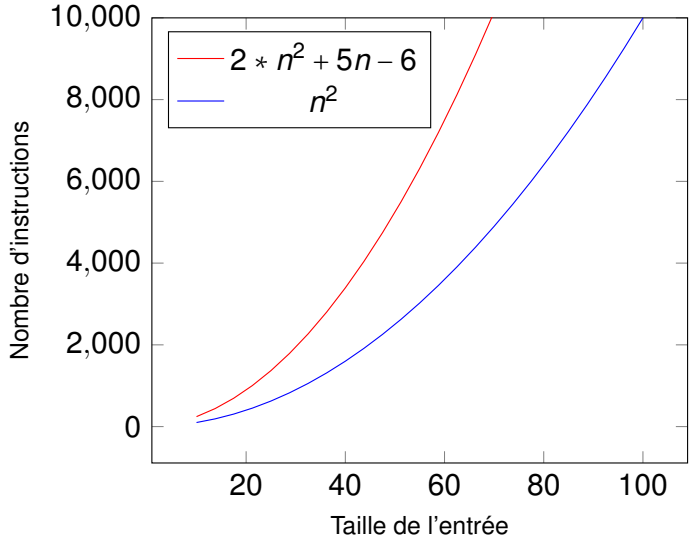
Les figures suivantes donnent la relation entre $T_{\text{Tri\_sél}}(n)$ et n^2.

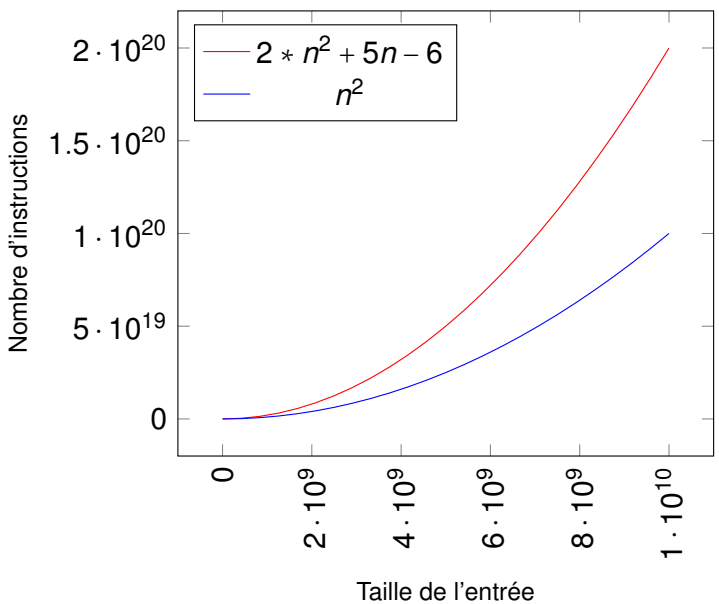


Grand *Oh*

Les figures suivantes donnent la relation entre $T_{\text{Tri\_sél}}(n)$ et n^2.







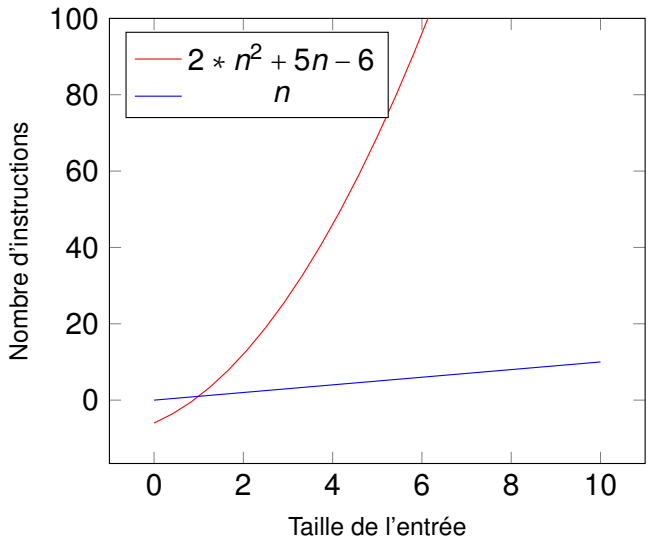
Grand *Oh*

Remarques

- Lorsque n est grand, alors le rapport entre $T_{\text{Tri\_sél}}(n)$ et n^2 est constant.
- Dans un même contexte (machine, langage, compilateur), un algorithme en n^2 sera toujours 2 fois plus rapide l'algorithme de tri par sélection.
- Ce rapport est rattrapé si l'algorithme de tri par sélection est implémenté sur une machine qui est 2 fois plus rapide que celle où il est implémenté l'algorithme en n^2.
- De ce fait, on considèrera que l'algorithme du tri par sélection et un algorithme en n^2 ont une même complexité calculatoire lorsque n est grand.

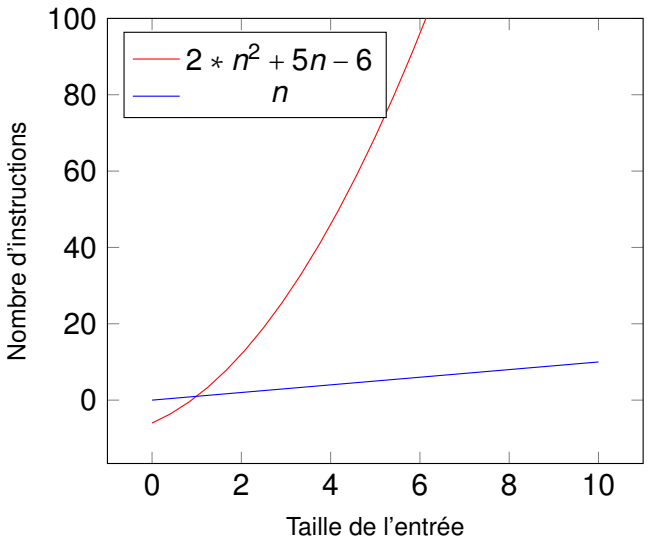
Grand *Oh*

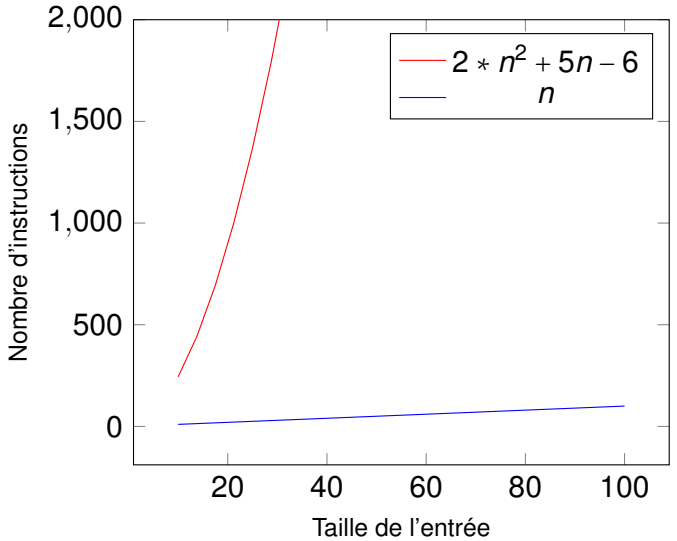
Les figures suivantes donnent la relation entre $T_{\text{Tri\_sél}}(n)$ et n.



Grand *Oh*

Les figures suivantes donnent la relation entre $T_{\text{Tri\_sél}}(n)$ et n.





- Lorsque n est grand, il est clair qu'un algorithme en n est plus efficace que l'algorithme de tri par sélection.
- Cette différence ne peut pas être rattrapée avec l'utilisation de machines plus puissantes.
- De ce fait, on considèrera que l'algorithme du tri par sélection n'est pas dans la même famille qu'un algorithme qui nécessite n instructions pour son exécution.

Grand *Oh*

Définition

Soit $T(n)$ le nombre d'instructions associé à un algorithme A. Soit $f(n)$ une fonction qui admet en paramètre n.
$T(n)$ est dite en $O(f(n))$ s'il existe deux constantes n_0 et k tel que:

$$\forall n \geq n_0, T(n) \leq k.f(n).$$

- Les constantes k et n_0 qui satisfont l'équation ci-dessus ne sont pas uniques.

Grand *Oh*

Intuitivement

$T(n)$ est dite en $O(f(n))$ si à partir d'un certain seuil n_0, le temps d'exécution de l'algorithme ne dépassera pas k fois $f(n)$.

Exemple

Reprenons notre fonction Tri_sélectif, où :

$$T_{\text{Tri\_sél}}(n) = 2n^2 + 5n - 6$$

Nous avons : $T_{\text{Tri\_sél}}(n) \in O(n^2)$.

- Soit $n_0 = 10$, $k = 3$.
- Il est facile de vérifier que

$$\forall n > n_0, T_{\text{Tri\_sél}}(n) \leq n^2.$$

Analyse de la complexité : Test de primalité

Rappel

Le nombre d'instructions élémentaires réalisées par la fonction de test de primalité est :

$$T_{Pr2}(n) = \begin{cases} 4 & \text{si n } \leq 2 \\ 5 + 3 * \sqrt{n} & \text{sinon} \end{cases}$$

Vérifions que $T_{Pr2}(n) \in O(\sqrt{n})$. Pour cela, prenons :

k=9 n_0=3

Il est facile de vérifier que :

$$\forall n > n_0, T_{Pr2}(n) \leq k.\sqrt{n}.$$

Propriétés de Oh : oublions les constantes

Constantes

- Rajouter à un algorithme un nombre constant d'instructions ne modifie pas sa complexité asymptotique : Si $T(n) \in O(f(n))$ et k est une constante, alors :

$$k + T(n) \in O(f(n)).$$

- Itérer un algorithme un nombre constant de fois ne modifie pas sa complexité asymptotique : Si $T(n) \in O(f(n))$ et k est une constante, alors :

$$k.T(n) \in O(f(n)).$$

Propriétés de Oh : garder la séquence la moins efficace

Addition

- Lorsqu'un algorithme est composé de deux séquences, alors sa complexité asymptotique revient à celle de la séquence la moins efficace : si $T_A(n) \in O(f(n))$ et $T_B(n) \in O(g(n))$ alors :

$$T_{A;B}(n) \in O(max(f(n), g(n))).$$

- Bien sûr, il est également vrai que :

$$T_{A;B}(n) \in O(f(n) + g(n)).$$

Exemple

SI $T(n)$ est sous la forme d'un polynôme

$$a_p.n^p + a_{p-1}.n^{p-1} + ... + a_1.n + a_0,$$

alors il suffit de garder le degré le plus élevé :

$$T(n) \in O(n^p).$$

Autres propriétés de Oh

Propriétés

- Transitivité : Si $T(n) \in O(f(n))$ et $f(n) \in O(g(n))$ alors

$$T(n) \in O(g(n)).$$

- Produit : Si $T_A(n) \in O(f(n))$ et $T_B(n) \in O(g(n))$ alors

$$T_A(n).T_B(n) \in O(f(n).g(n)).$$

Le produit reflète la présente de boucles imbriquées.

Ω *et* Θ

Grand Ω

Définition

Soit $T(n)$ le nombre d'instructions associé à un algorithme A. Soit $f(n)$ une fonction qui admet en paramètre n.
$T(n)$ est dite en $\Omega(f(n))$ s'il existe deux constantes n_0 et k tel que:

$$\forall n \geq n_0, T(n) \geq k.f(n).$$

Grand Ω : Test de primalité

Rappel

Rappelons que le nombre d'instructions élémentaires réalisées par la fonction de test de primalité est de :

$$T_{Pr2}(n) = \begin{cases} 4 & \text{si n } \leq 2 \\ 5 + 3 * \sqrt{n} & \text{sinon} \end{cases}$$

Vérifions que $T_{Pr2}(n) \in \Omega(\sqrt{n})$. Pour cela, prenons :

k=2 n_0=3

Il est facile de vérifier que :

$$\forall n > n_0, T_{Pr2}(n) \geq k.\sqrt{n}.$$

Grand Θ

Définition

Soit $T(n)$ le nombre d'instructions associé à un algorithme A. Soit $f(n)$ une fonction qui admet en paramètre n.
$T(n)$ est dite en $\Theta(f(n))$ si :

$$T(n) \in O(f(n)) \text{ et } T(n) \in \Omega(f(n))$$

- Dit autrement, $T(n)$ est dite en $\Theta(f(n))$ s'il existe trois constantes n_0, k et c tel que:

$$\forall n \geq n_0, c.f(n) \leq T(n) \leq k.f(n).$$

Grand Θ : Test de primalité

Rappel

Rappelons de nouveau que le nombre d'instructions élémentaires réalisées par la fonction de test de primalité est de :

$$T_{Pr2}(n) = \begin{cases} 4 & \text{si n } \leq 2 \\ 5 + 3 * \sqrt{n} & \text{sinon} \end{cases}$$

Comme nous avons établi que :

$$T_{Pr2}(n) \in \Omega(\sqrt{n}) \text{ et } T_{Pr2}(n) \in O(\sqrt{n})$$

Nous pouvons affirmer que :

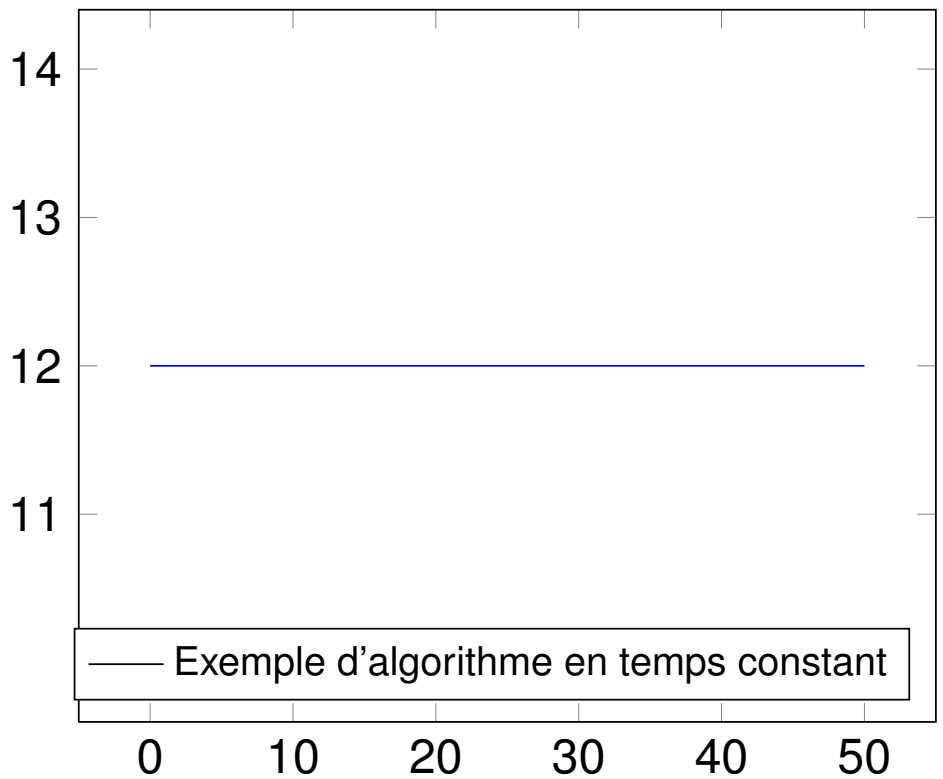
$$T_{Pr2}(n) \in \Theta(\sqrt{n})$$

Classes remarquables de fonctions de complexité temporelle

Temps constant $Oh(1)$

Temps constant

La complexité d'un algorithme est dite en temps constant si le temps d'exécution de l'algorithme reste inchangé et est indépendante de la taille de l'entrée.



Temps constant

Remarques

- Les algorithmes en $O(1)$ sont les plus efficaces.
- Toute suite d'instructions élémentaires est calculée en $O(1)$.
- Pour dire qu'un algorithme s'exécute en $f(n)$ plus un nombre d'instructions élémentaires, on écrit souvent $f(n) + O(1)$

Exemple

Ecrire une fonction qui prend en entrée un tableau de chaînes de caractères, représentant les noms des élèves de L3 informatique, et retourne de manière aléatoire le nom du volontaire (pour écrire cette fonction).

Algorithmes en temps constant : exemple

```
char *volontaire (char *EtudiantsL3[], int Taille)
{
  return (EtudiantsL3 [((int) rand ()) % Taille]);
}
```

Analyse de la complexité temporelle de la fonction *volontaire*

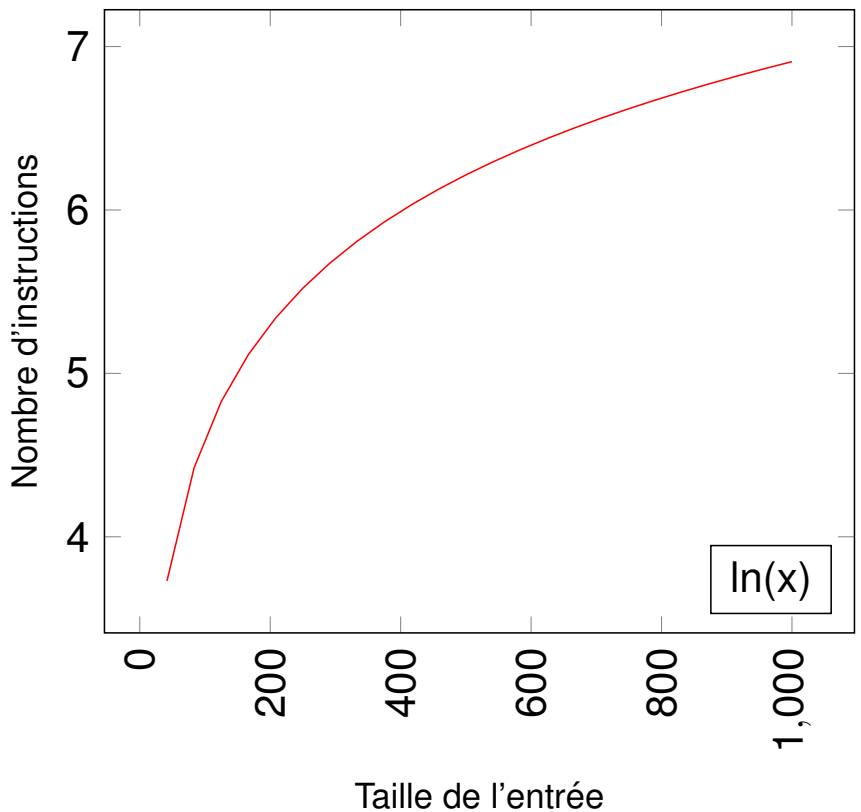
Cette fonction est en $O(1)$:

- Elle exécute un certain nombre d'instructions élémentaires :
  - Appel de la fonction rand() (qui se fait en $O(1)$)
  - le reste de la division modulaire (%)
  - le retour du résultat de la fonction
- Cette fonction ne dépend pas de la taille d'entrée.

Temps logarithmique

Temps logarithmique

Un algorithme qui s'exécute en un temps logarithmétique est noté $O(log_2(n))$.



Temps logarithmique

Remarques

- Un algorithme en $O(log_2(n)$ est extrêmement efficace. Il accepte des entrées de tailles très grandes.
- La base n'a aucune importance, puisque pour deux bases données a,b, $log_a(n)$ est proportionnelle à $log_a(n)$ (c'est-à-dire $log_a(n) \propto log_b(n)$).
- Rappel :

$$log_a(n) = \frac{log_b(n)}{log_b(a)}.$$

Temps logarithmique

Remarques

- Les algorithmes en $O(log_2(n))$ ou de la forme $O(log_2(n) * f(n))$ reflète souvent l'idée de *diviser pour régner* :
  - on décompose un problème p en sous-problèmes $p_1, ..., p_m$,
  - chacun de ces problèmes p_i est de nouveau décomposé en sous-problèmes $p_{i1}, ..., p_{im}$
  - on s'arrête lorsque les sous-problèmes se traitent en temps constant (en O(1)).
- L'entrée ou le paramètre n décroit en $\frac{n}{m}$, $\frac{n}{m^2}$, ..., 1 où m est le nombre de de sous-problèmes associé à un problème.

Temps logarithmique

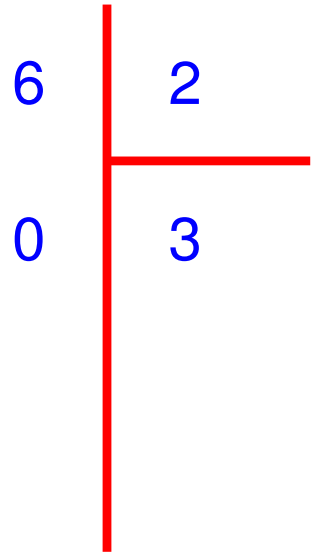
Exemple

Ecrire une fonction qui :

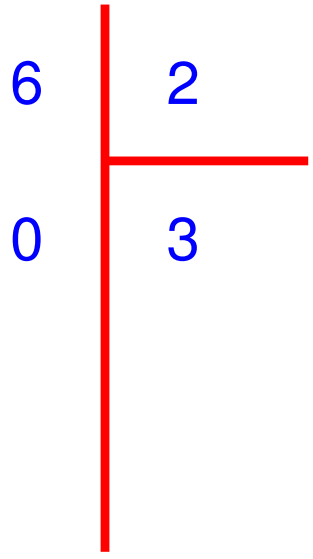
- prend en entrée un entier positif n,
- calcule sa décomposition binaire,
- la range dans un tableau et
- l'imprime.

Par exemple, si $n = 7$ la fonction affichera "111".
On utilise l'algorithme basé sur la sauvegarde des restes des divisions successives par 2 du nombre initial.

Exemple : Conversion en binaire

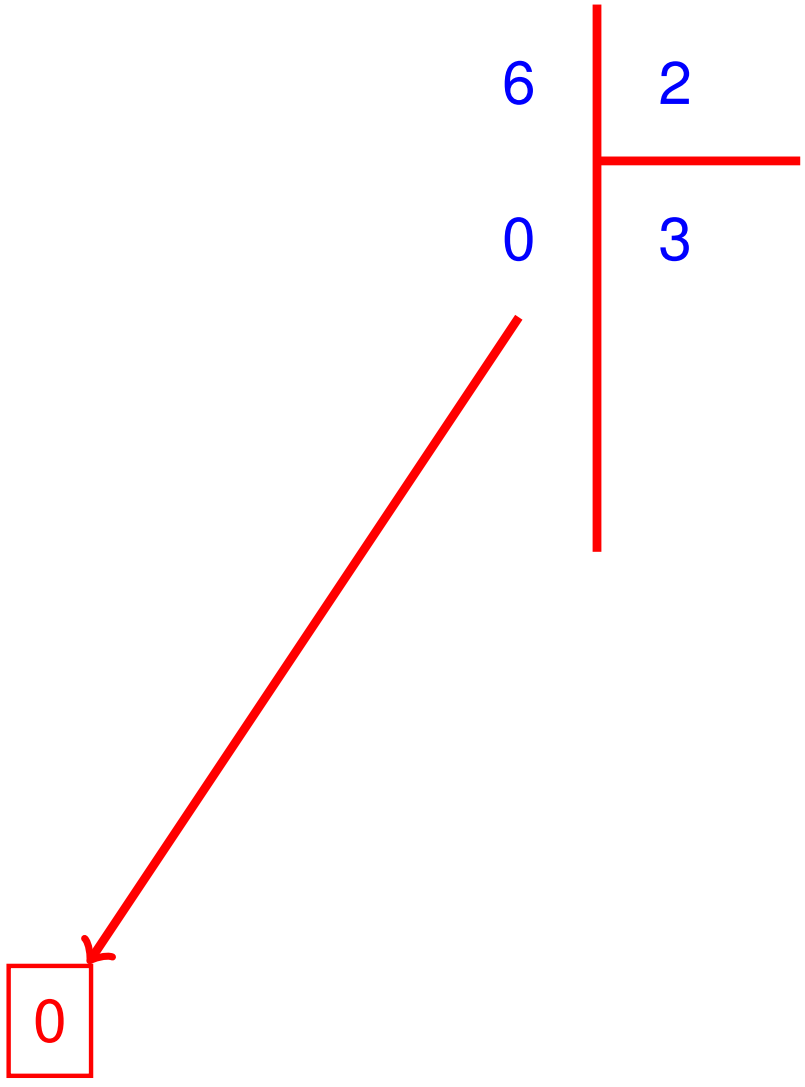


Exemple : Conversion en binaire

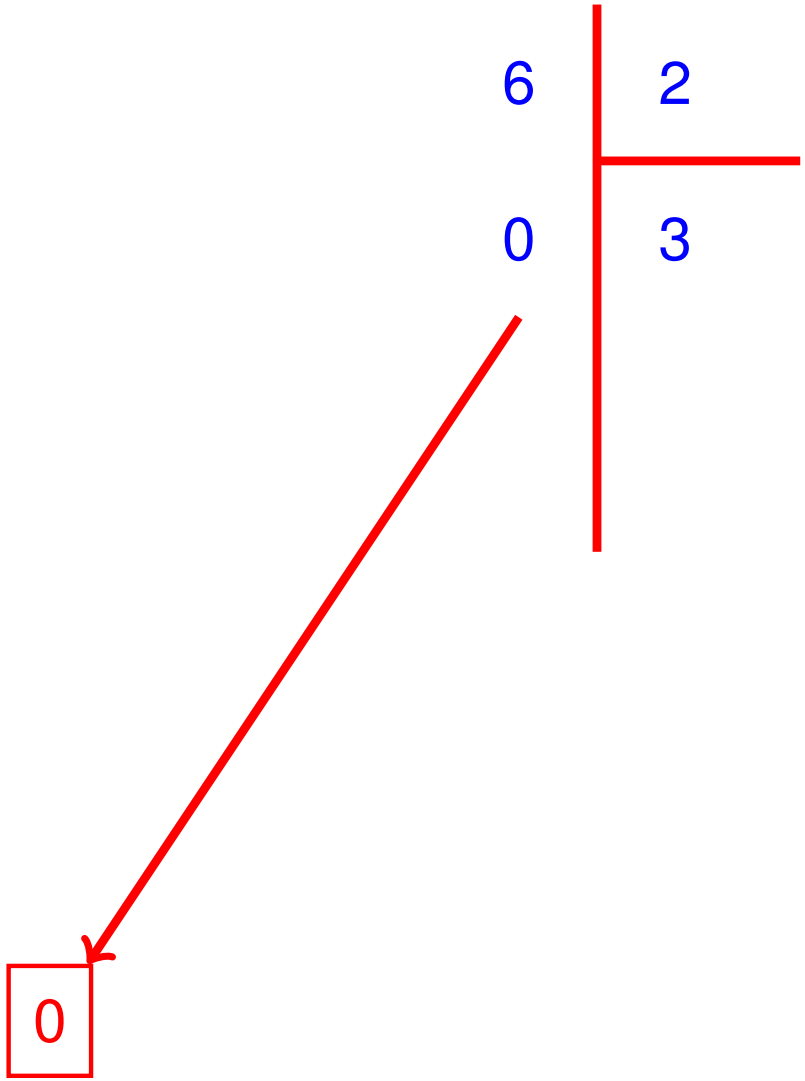


0

Exemple : Conversion en binaire

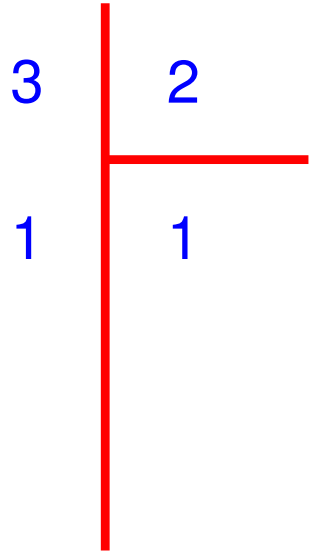


Exemple : Conversion en binaire



Reste après une division par 2 du nombre 6.

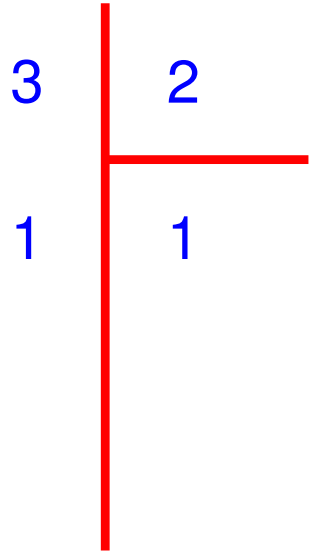
Exemple : Conversion en binaire



0

Rappel : reste après 1 division par 2 du nombre 6.

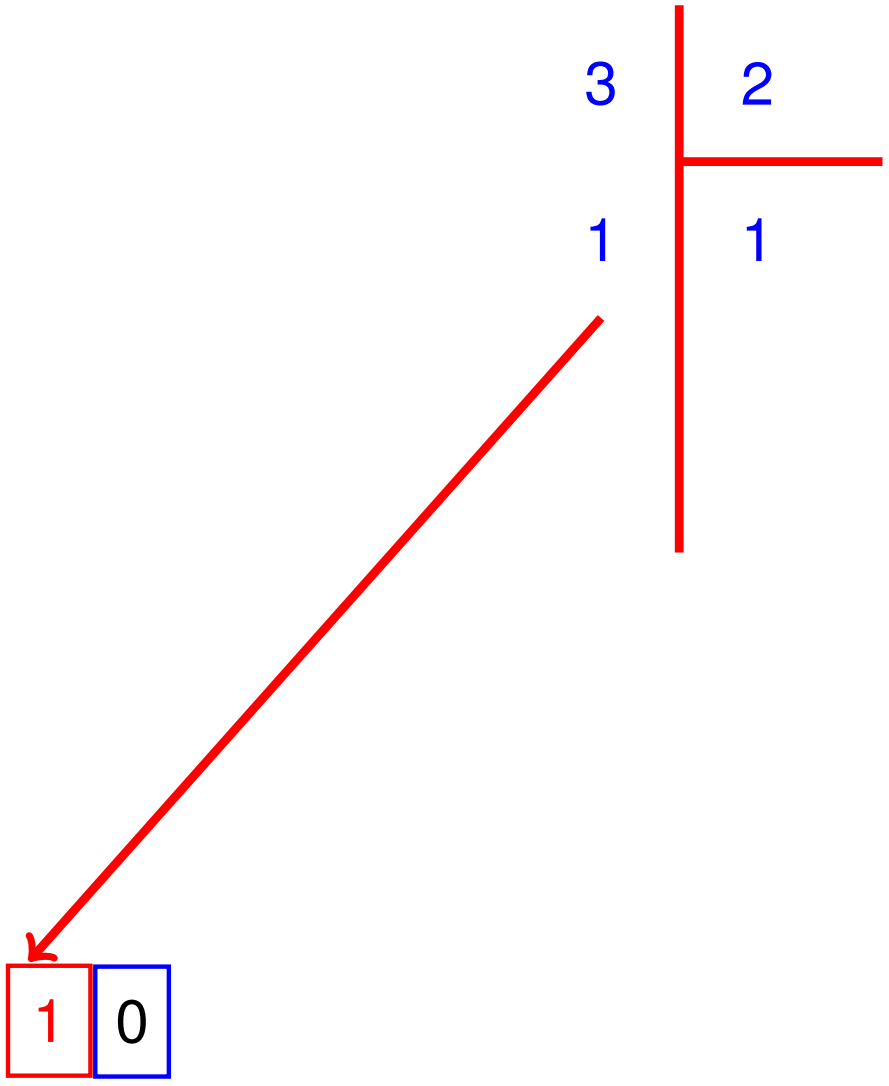
Exemple : Conversion en binaire



0

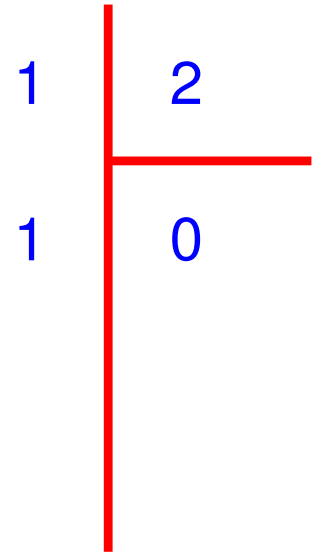
Rappel : reste après 1 division par 2 du nombre 6.

Exemple : Conversion en binaire



Rappel : reste après 1 division par 2 du nombre 6.

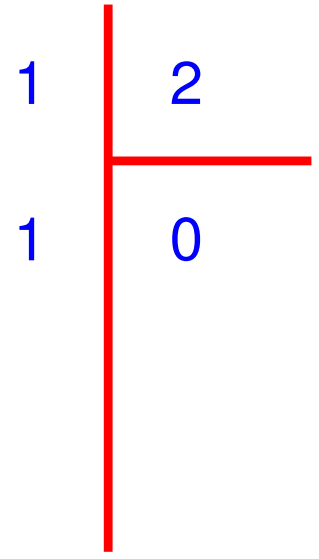
Exemple : Conversion en binaire



1	0

Rappel : liste des restes après 2 divisions successives par 2 du nombre 6.

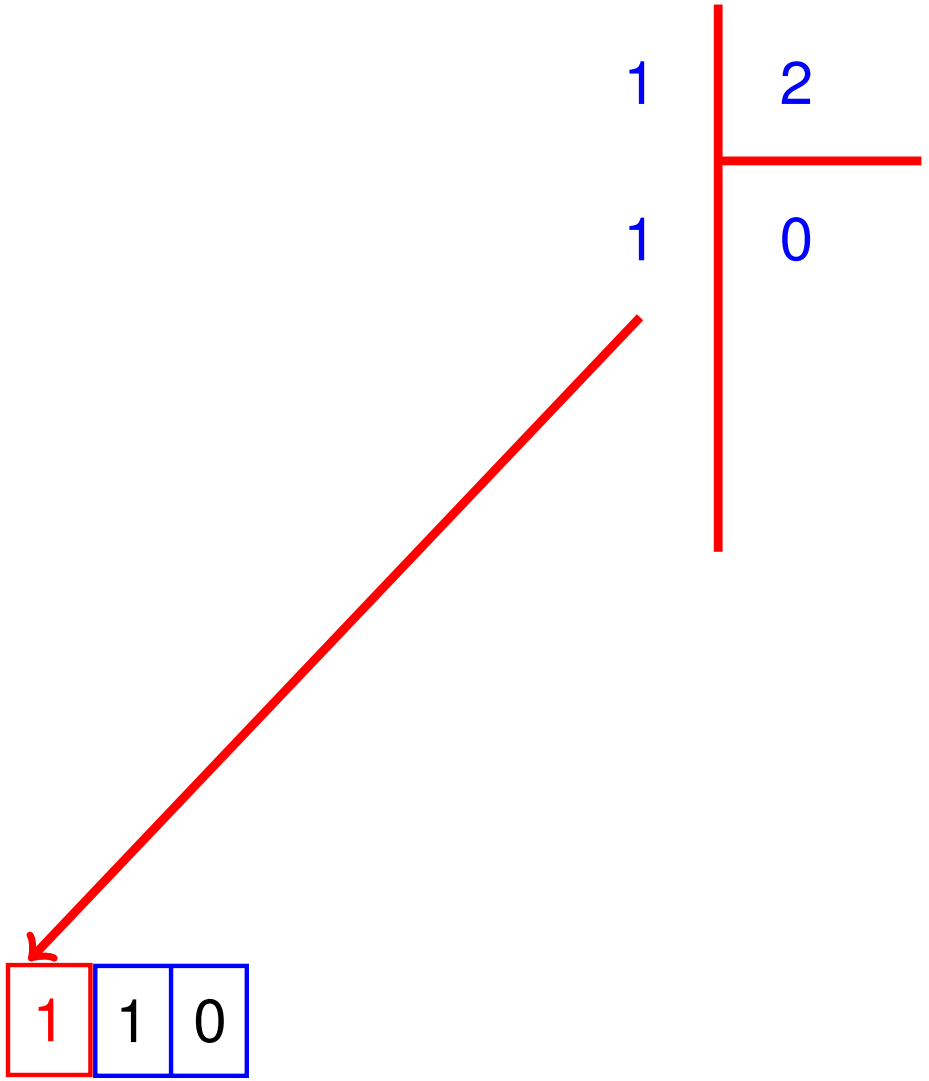
Exemple : Conversion en binaire



1	0

Rappel : liste des restes après 2 divisions successives par 2 du nombre 6.

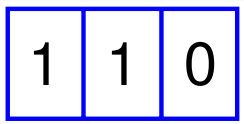
Exemple : Conversion en binaire



Rappel : liste des restes après 2 divisions successives par 2 du nombre 6.

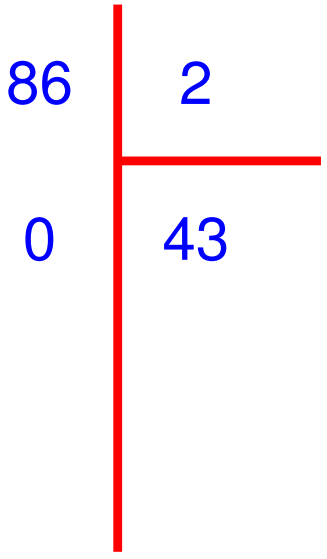
Exemple : Conversion en binaire

La conversion binaire du nombre 6 donne :

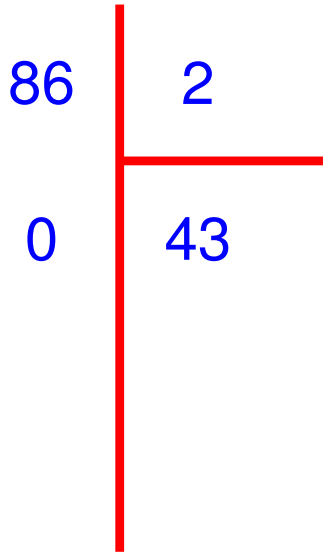


Un autre exemple

Exemple : Conversion en binaire

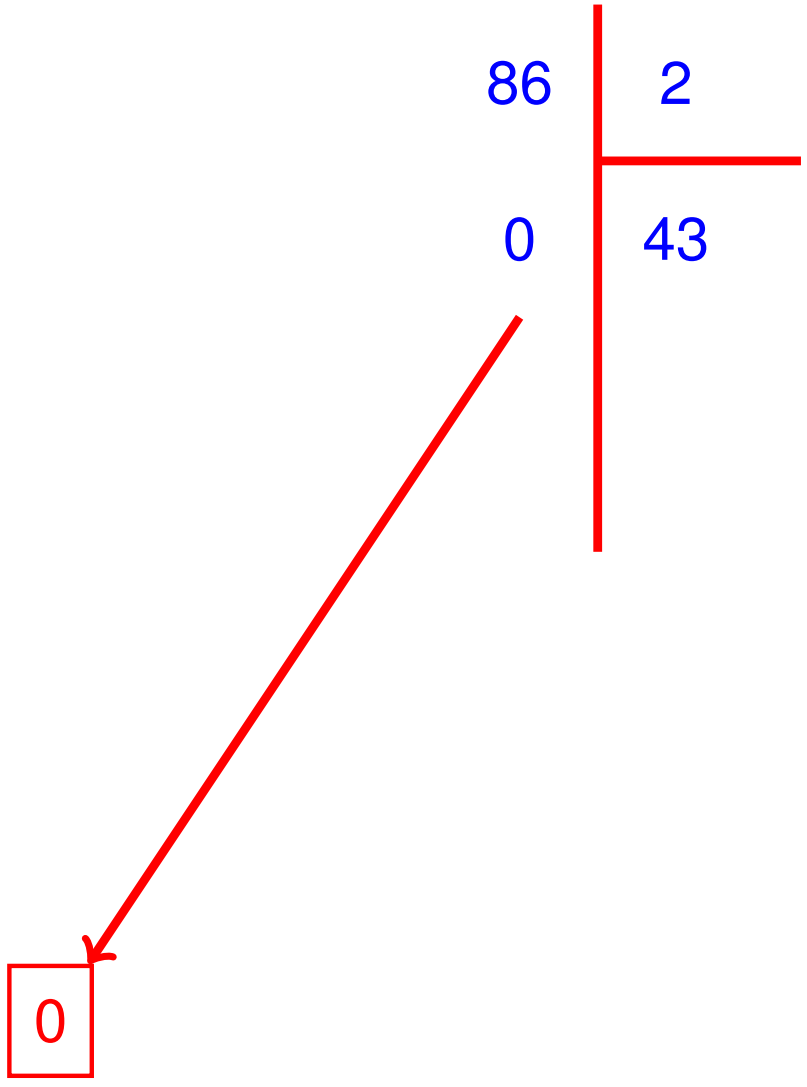


Exemple : Conversion en binaire

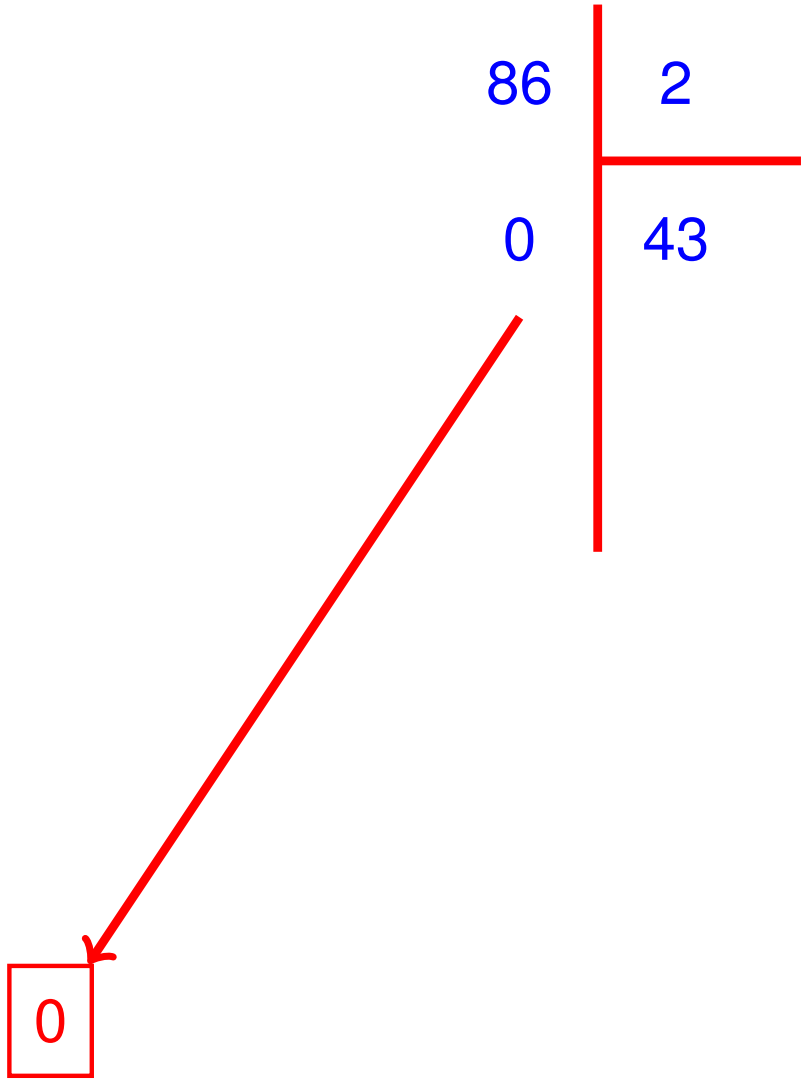


0

Exemple : Conversion en binaire

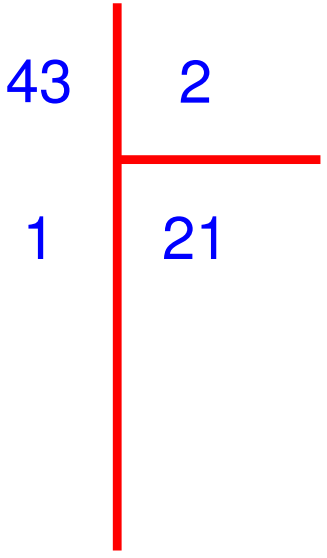


Exemple : Conversion en binaire



Reste après une division par 2 du nombre 86.

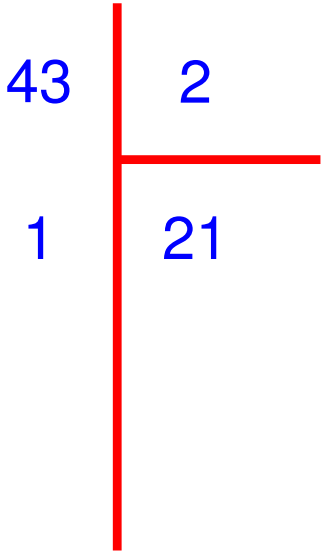
Exemple : Conversion en binaire



0

Rappel : reste après 1 division par 2 du nombre 86.

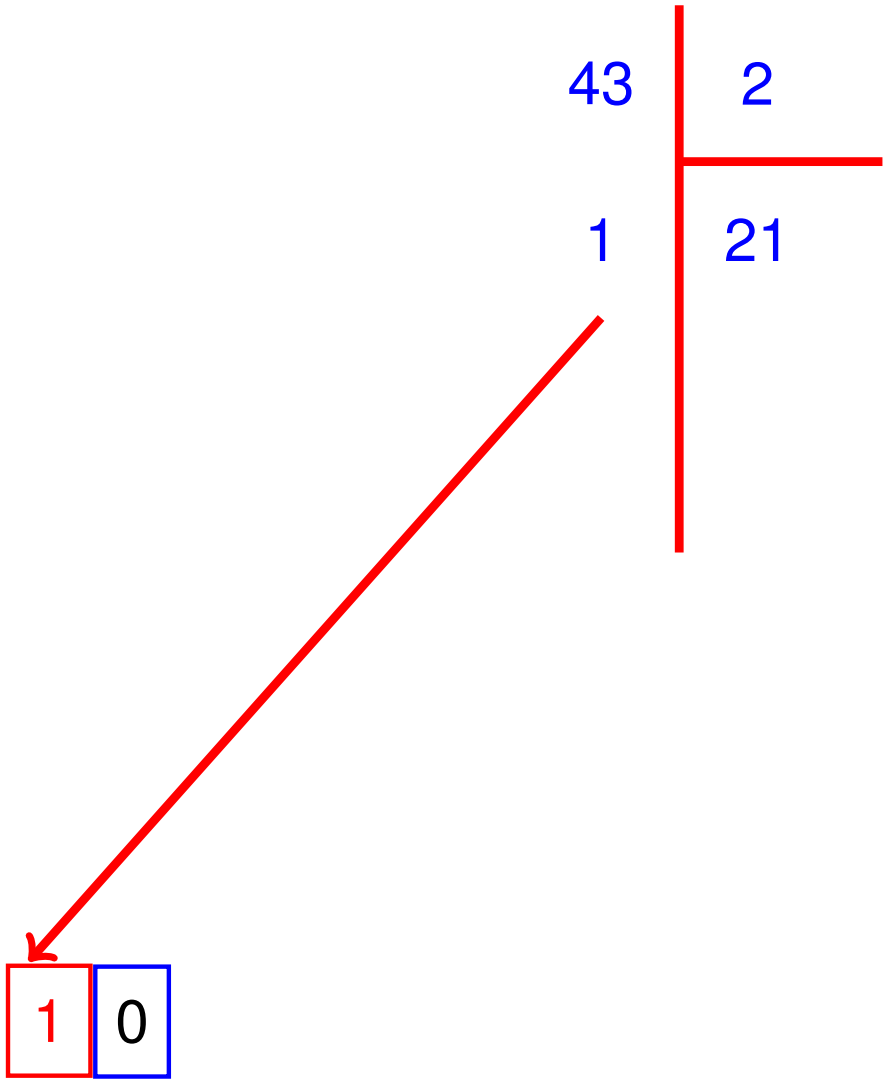
Exemple : Conversion en binaire



0

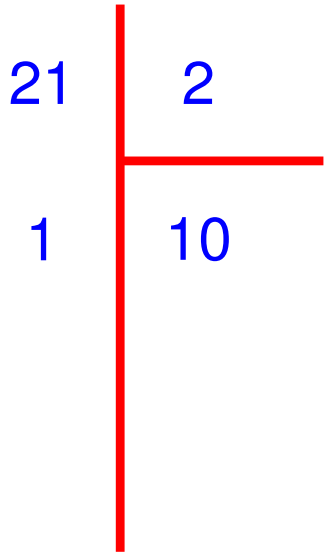
Rappel : reste après 1 division par 2 du nombre 86.

Exemple : Conversion en binaire



Rappel : reste après 1 division par 2 du nombre 86.

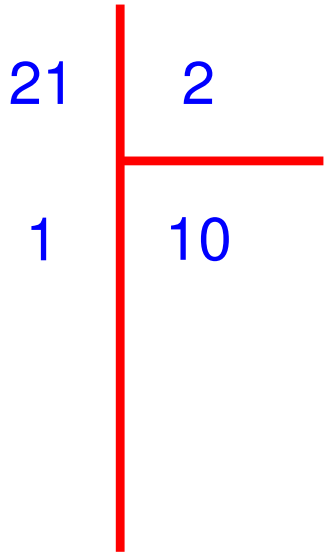
Exemple : Conversion en binaire



1	0

Rappel : liste des restes après 2 divisions successives par 2 du nombre 86.

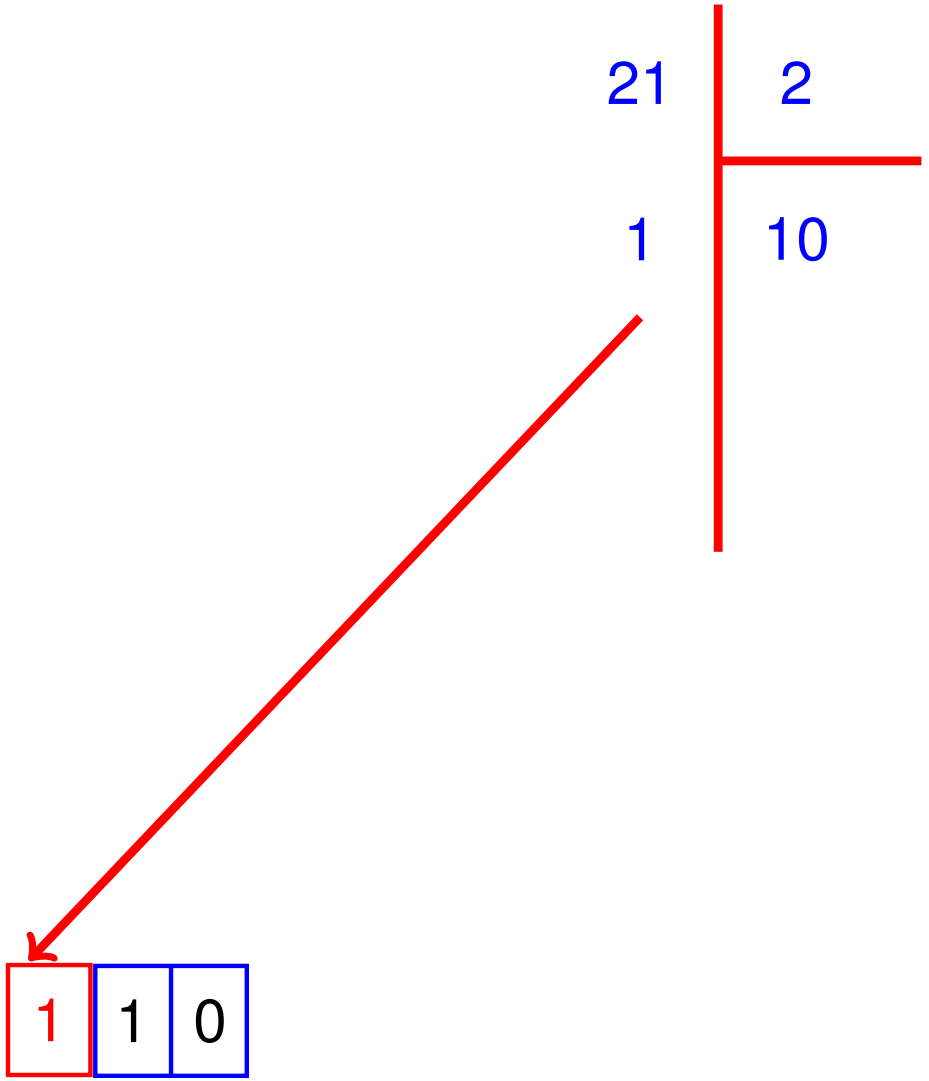
Exemple : Conversion en binaire



1	0

Rappel : liste des restes après 2 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire



Rappel : liste des restes après 2 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire

10	2
0	5

1	1	0

Rappel : liste des restes après 3 divisions successives par 2 du nombre 86.

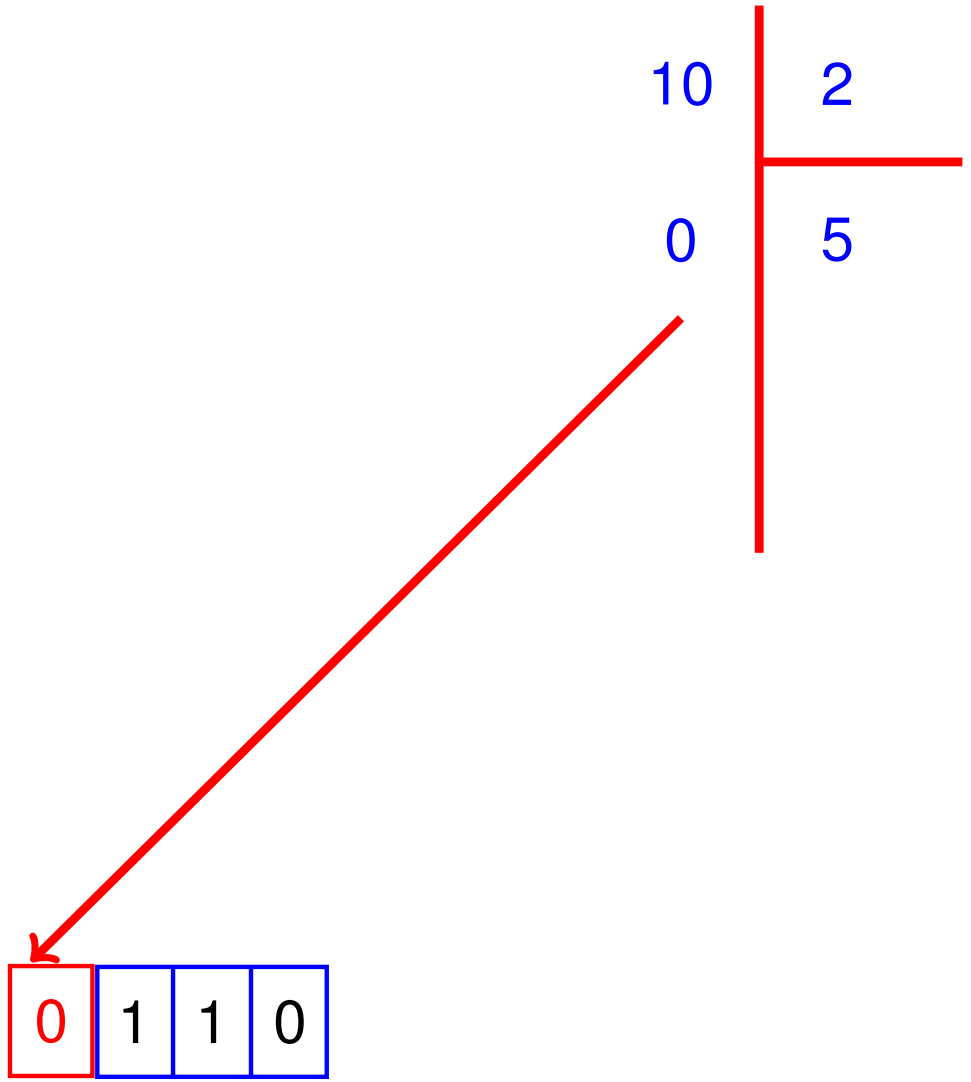
Exemple : Conversion en binaire

10	2
0	5

1	1	0

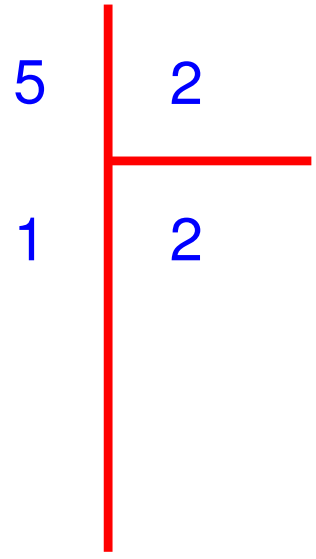
Rappel : liste des restes après 3 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire



Rappel : liste des restes après 3 divisions successives par 2 du nombre 86.

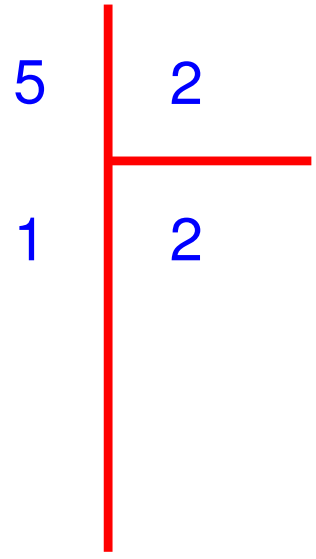
Exemple : Conversion en binaire



0	1	1	0

Rappel : liste des restes après 4 divisions successives par 2 du nombre 86.

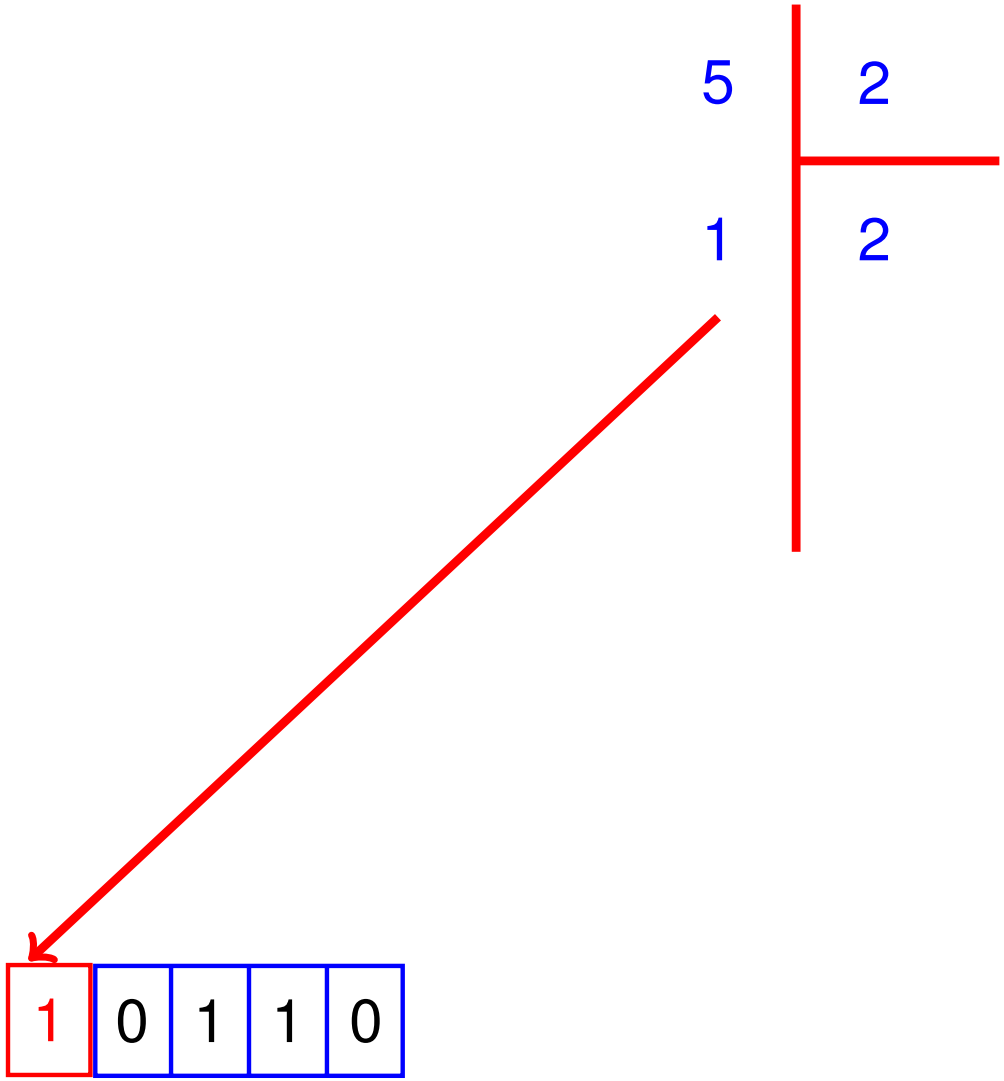
Exemple : Conversion en binaire



0	1	1	0

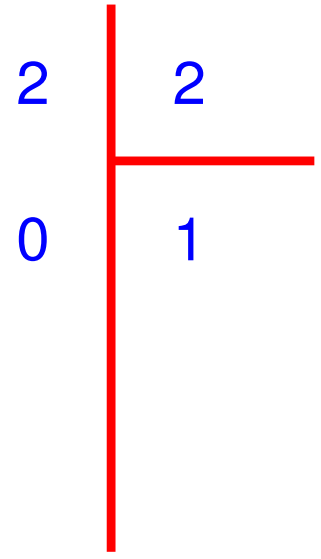
Rappel : liste des restes après 4 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire



Rappel : liste des restes après 4 divisions successives par 2 du nombre 86.

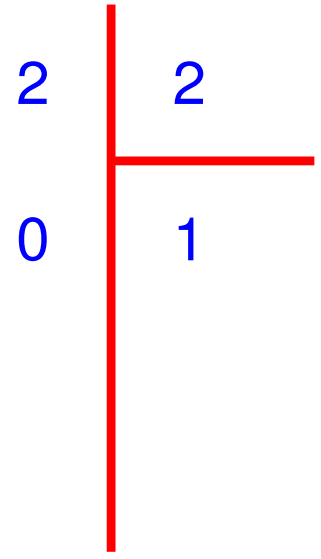
Exemple : Conversion en binaire



1	0	1	1	0

Rappel : liste des restes après 5 divisions successives par 2 du nombre 86.

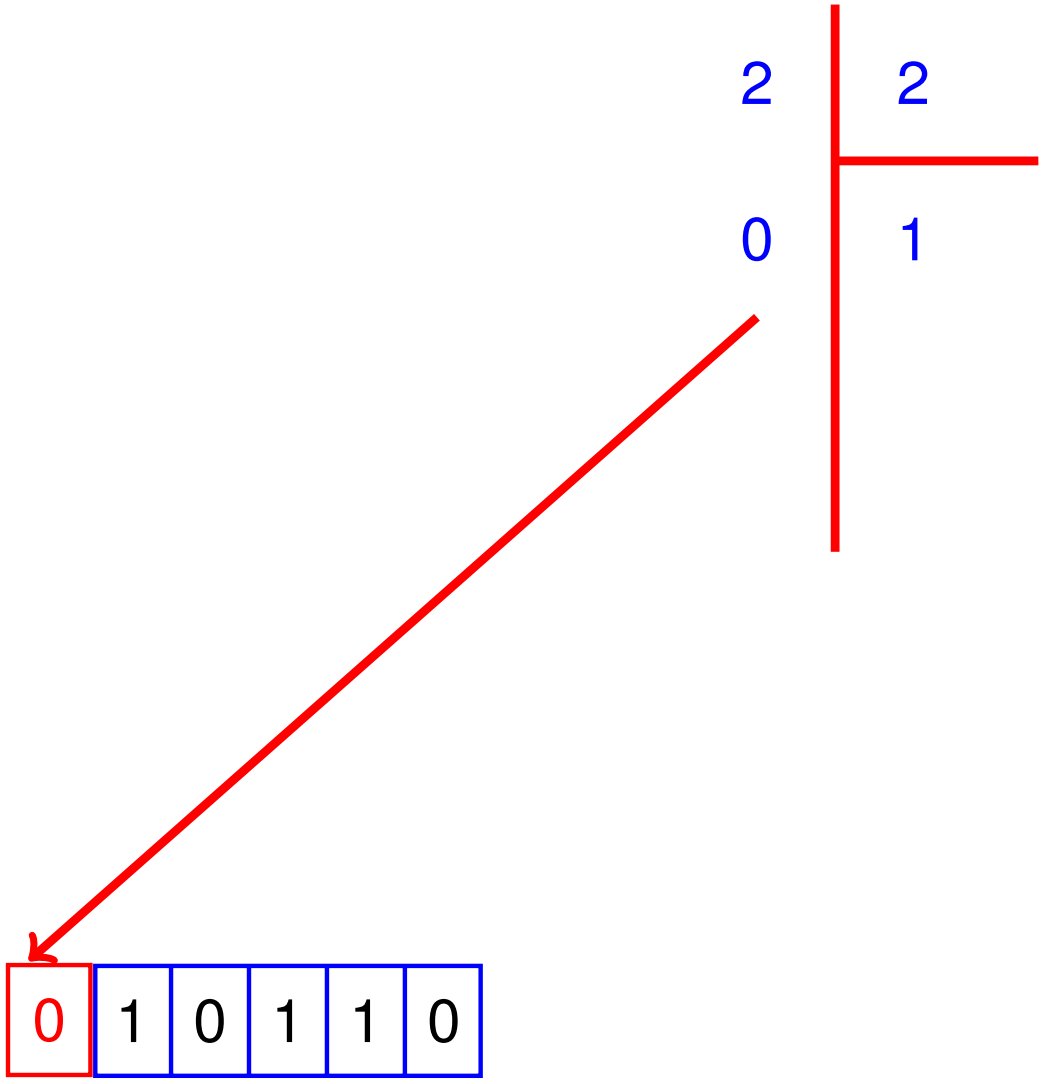
Exemple : Conversion en binaire



1	0	1	1	0

Rappel : liste des restes après 5 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire



Rappel : liste des restes après 5 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire

1	2
1	0

0	1	0	1	1	0

Rappel : liste des restes après 6 divisions successives par 2 du nombre 86.

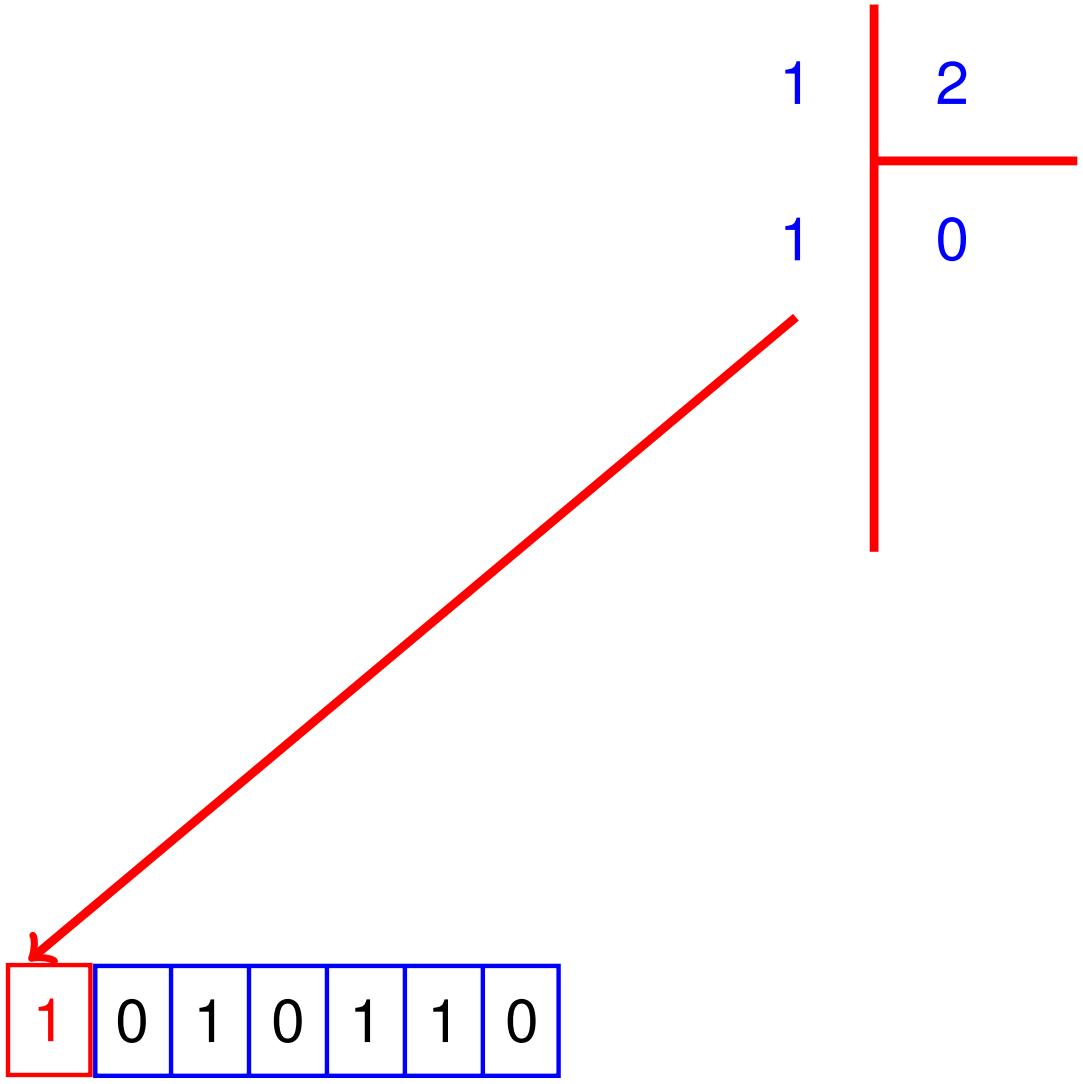
Exemple : Conversion en binaire

1	2
1	0

0	1	0	1	1	0

Rappel : liste des restes après 6 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire



Rappel : liste des restes après 6 divisions successives par 2 du nombre 86.

Exemple : Conversion en binaire

La conversion binaire du nombre 86 donne :

1	0	1	0	1	1	0

Temps logarithmique : exemple (conversion en binaire)

```
void decomposition (const int n)
{
  printf ("La decomposition binaire de %d est :", n);
  if (n==0) printf ("0\n");
  else
  {
  unsigned char *binaire;
  binaire=malloc ((int) (4*log10(n))*sizeof(unsigned char));
  int i=0; int m=n;
  while (m>0)
  {
    binaire[i]=m%2;
    m=m/2;
    i++;
  }
  for (m=i-1;m>=0;m--) printf("%d",binaire[m]);
  printf("\n");
  }
}
```

Temps logarithmique : exemple (conversion en binaire)

La fonction décomposition contient

- au plus 6 instructions élémentaires :

```
1   printf ("La decomposition binaire de %d est :", n);
2   if (n==0)      printf ("0\n");
3   binaire=malloc ((int) (4*log10(n))*sizeof(unsigned char));
4   int i=0;
5   int m=n;
6   printf("\n");
```

- Deux boucles : une boucle while et une boucle for

Temps logarithmique : exemple (conversion en binaire)

- la boucle ”while”

```
1   while (m>0){
2     binaire[i]=m%2;
3     m=m/2;
4     i++;  }
```

réalise 4 instructions élémentaires à chaque itération.

- Il reste à calculer le nombre d'itérations de la boucle while :
  - La valeur m (initialisée à n) décroit de moitié à chaque itération
  - Au fait, la valeur i donne le nombre d'itérations et calcule le nombre de fois $m = n$ a été divisé par 2.
  - Lorsque $m = 1$ (avant dernière itération) alors nous avons :
    $$\frac{n}{2^i} = 1 \quad \text{c'est-à-dire :} \quad i = log_2(n)$$
  - Le nombre d'instructions réalisées par la boucle while est de : $4 * (log_2(n) + 1)$

Temps logarithmique : exemple (conversion en binaire)

- la dernière boucle (la boucle ”for”)

```
1   for (m=i-1;m>=0;m--) printf("%d",binaire[m]);
```

réalise

- 1 instruction d'initialisation (m=i-1;).
- A chaque itération (m=$log_2(n)$ jusqu'à 0) 3 instructions élémentaires (une comparaison, une décrémentation et une impression) sont réalisées.
- Le nombre d'instructions réalisées par la boucle ”for” est de :

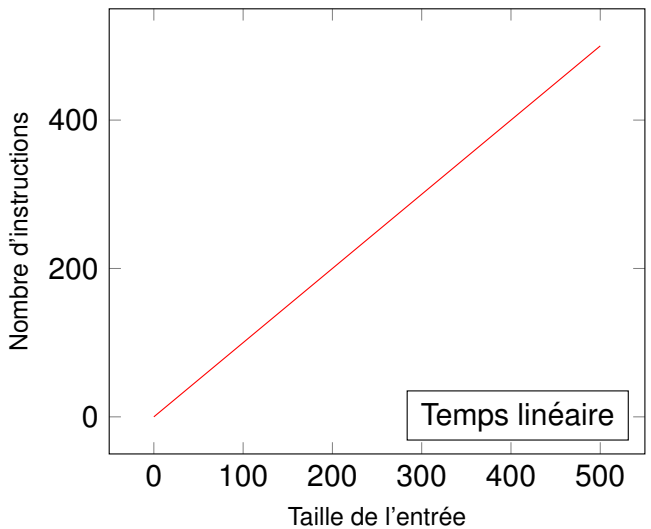
$$3 * (log_2(n) + 1) + 1.$$

Résultat final de la complexité de décomposition

$$\begin{array}{lcccccc} T_{\text{décomposition}}(n) & = & T_{\text{boucle while}}(n) & + & T_{\text{boucle for}}(n) & + & 6 \\ & = & 4 * (log_2(n) + 1) & + & 3 * (log_2(n) + 1) + 1 & + & 6 \\ & \in & O(log_2(n)) & & & & \end{array}$$

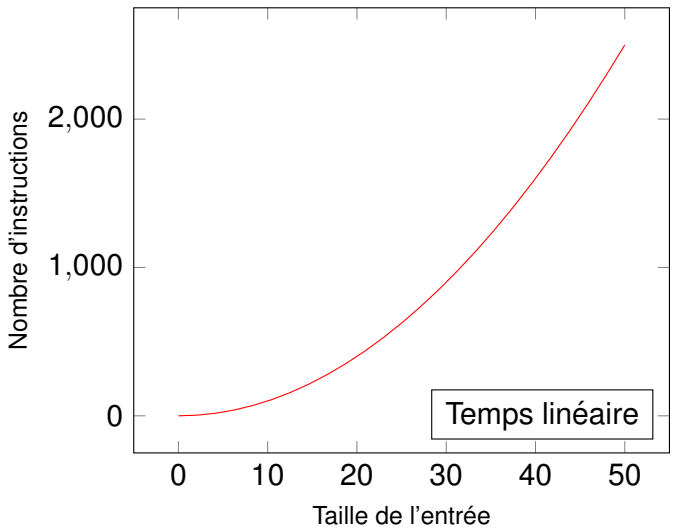
Temps linéaire

- Un algorithme qui s'exécute en un temps linéaire est noté $O(n)$.
- Il indique qu'à partir d'une certaine entrée, la complexité de l'algorithme est plus petite qu'un algorithme qui réalise un nombre constant d'itérations de taille n.
- Un exemple simple d'algorithme en $O(n)$ est la recherche d'un élément dans un tableau non-trié de taille n.



Temps quadratique

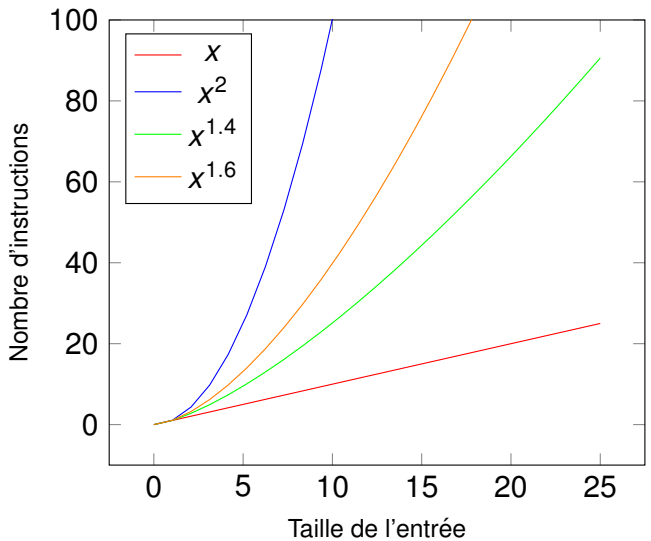
- Un algorithme qui s'exécute en un temps quadratique est noté $O(n^2)$.
- Il indique qu'à partir d'une certaine entrée, la complexité de l'algorithme est plus petite qu'un algorithme qui réalise un nombre constant de 2 boucles imbriquées chacune de taille n.
- Un exemple simple d'algorithme en $O(n^2)$ est l'addition de deux matrices carrées chacune de taille nxn.



Entre $O(n)$ et $O(n^2)$

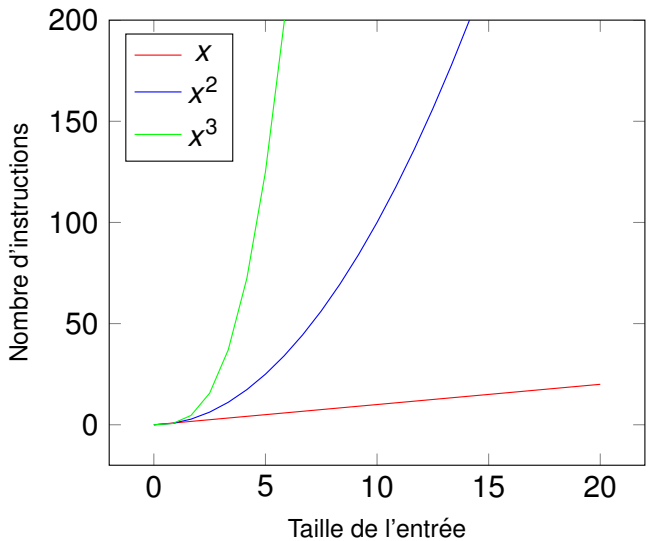
Remarque

- Les algorithmes en $O(n^r)$ avec $1 < r < 2$ sont strictement plus efficaces que n^2 et strictement moins efficaces que n
- Un exemple avec $r = 1.5$ qui représente un algorithme en racine carrée fois n, c'est-à-dire $O(n * \sqrt{(n)})$.



Temps polynomial

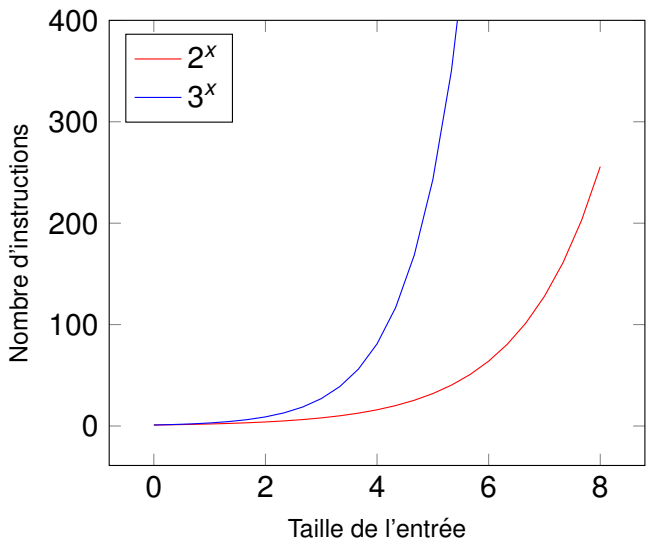
- Un algorithme qui s'exécute en un temps polynomial est noté $O(n^c)$, où c est une constante.
- Surtout il ne faut pas confondre entre $O(n^c)$ et $O(c^n)$ (qui est exponnentiel).
- Les algorithmes en temps polynomial sont considérés comme efficaces.



Temps exponnentiel

Temps exponnentiel

- Un algorithme qui s'exécute en un temps exponentiel est noté $O(c^n)$, où c est une constante.
- Les algorithmes en temps exponentiel ne sont pas efficaces.
- Attention si c_1 et c_2 sont deux constantes différentes alors un algorithme en $O(c_1^n)$ se comporte différemment d'un algorithme en $O(c_2^n)$ (ils n'ont pas la même complexité).



Oh, Ω, Θ : que choisir?

Une question de précision

- L'idéal est d'utiliser Θ qui donne une bonne précision du comportement de la complexité d'un algorithme. A défaut, il faut simultanément utiliser grand *Oh* et Ω.
- L'inconvénient de la notation *Oh* est que si vous établissez que votre algorithme est en :

$$O(n^2)$$

alors on peut toujours confirmer qu'il est également en

$$O(n^3), O(log_2(n) * n^3), O(5^n), \text{ etc.}$$

toutes les classes de complexité qui sont moins efficaces que $O(n^2)$.

Oh, Ω, Θ : que choisir?

Une question de précision

- De manière similaire, si votre algorithme est en

$$\Omega(n^2)$$

alors il est aussi en

$$\Omega(n), \Omega(n^{0.24}), \Omega(log_2(n)), \Omega(1) \text{ etc.}$$

toutes les classes de complexité qui sont plus efficaces que $\Omega(n^2)$.

Oh, Ω, Θ : que choisir?

En complexité temporelle, par défaut (en pratique):

- On se situe dans le pire des cas. Si on fait une analyse en moyenne ou en situations favorables, on le précise explicitement.
- On utilise la notation grand *Oh* avec, dans la mesure du possible, des fonctions *Oh* proches de Θ. Par exemple, si le nombre d'instructions d'un algorithme est $T(n) = 3 * n^2 + 4$, on dit que $T(n) \in O(n^2)$ et non $T(n) \in O(n^3)$ même si les deux assertions sont justes. La première assertion est plus précise que la deuxième.

Oh, Ω, Θ : que choisir?

- Pour certains problèmes il est difficile d'établir de manière précise la complexité d'un algorithme. Dans ce cas, la recherche consiste à encadrer la complexité de cet algorithme par deux entités $O(f(n))$ et $\Omega(g(n))$.

Oh, Ω, Θ : que choisir?

Exemple

Evaluons la complexité de la fonction ”nombre de nombres premiers compris entre 1 et n” avec un test de la primalité qui s’arrête à la racine carrée de n.

Nombre de nombres premiers entre 1 et n

```
1  int Nbredenbrespremiersf(const  int n)
2  {
3   int i, nbrepremiers=0;
4   for(i = 1; i <= n; i++)
5   {
6   if (Estpremierracine2(i) == 1)
7    nbrepremiers++;
8   }
9   return nbrepremiers;
10 }
```

Nombre de nombres premiers entre 1 et n

- En plus du coût de la boucle, nous avons 2 instructions élémentaires :

```
1   nbrepremiers=0;          // affectation
2   return nbrepremiers; //retour de valeur
```

- Pour la boucle :

```
1          for(i = 1; i <= n; i++){
2           if (Estpremierracine2(i) == 1)
3             nbrepremiers++; }
```

nous exécutons d'abord une (1) instruction d'initialisation : i=1.

Nombre de nombres premiers entre 1 et n

- Pour chaque itération i de la boucle (de 1 jusqu'à n), nous réalisons (dans le pire des cas) (4) opérations élémentaires :
  - une comparaison : $i <= n$;
  - une incrémentation : $i++$;
  - une comparaison : Estpremierracine2(i) == 1
  - une incrémentation : nbrepremiers++;
- et un appel à la fonction Estpremierracine2(i) qui coûte

$$T_{Pr2}(i) = 5 + 3 * \sqrt{i}$$

pour $n > 2$ (et 4 sinon).

Nombre de nombres premiers entre 1 et n

Calcul final

Le nombre d'instructions nécessaires à l'exécution de la fonction "nombre de nombre premiers entre 1 et n" est de :

$$T_{\text{nprem}}(n) = 3 + ((4 + T_{\text{Pr2}}(1)) + (4 + T_{\text{Pr2}}(2)) + \ldots + (4 + T_{\text{Pr2}}(n)))$$

Nombre de nombres premiers entre 1 et n

Calcul final

Le nombre d'instructions nécessaires à l'exécution de la fonction "nombre de nombre premiers entre 1 et n" est de :

$$
\begin{aligned}
&T_{\text{nprem}}(n) \\
&= 3 + ((4 + T_{\text{Pr2}}(1)) + (4 + T_{\text{Pr2}}(2)) + \ldots + (4 + T_{\text{Pr2}}(n)) \\
&= 4*n + 3 + (T_{\text{Pr2}}(1) + T_{\text{Pr2}}(2) + \ldots + T_{\text{Pr2}}(n))
\end{aligned}
$$

Nombre de nombres premiers entre 1 et n

Calcul final

Le nombre d'instructions nécessaires à l'exécution de la fonction "nombre de nombre premiers entre 1 et n" est de :

$$
\begin{aligned}
& T_{\text{nprem}}(n) \\
&= 3 + ((4 + T_{\text{Pr2}}(1)) + (4 + T_{\text{Pr2}}(2)) + \ldots + (4 + T_{\text{Pr2}}(n)) \\
&= 4*n + 3 + (T_{\text{Pr2}}(1) + T_{\text{Pr2}}(2) + \ldots + T_{\text{Pr2}}(n)) \\
&= 4*n + 3 + (4+4 + T_{\text{Pr2}}(3) + \ldots + T_{\text{Pr2}}(n))
\end{aligned}
$$

Nombre de nombres premiers entre 1 et n

Calcul final

Le nombre d'instructions nécessaires à l'exécution de la fonction "nombre de nombre premiers entre 1 et n" est de :

$T_{\text{nprem}}(n)$

$$
\begin{array}{lllllllll}
= 3 & + & ((4 + T_{\text{Pr2}}(1)) & + & (4 + T_{\text{Pr2}}(2)) & + & \ldots & + & (4 + T_{\text{Pr2}}(n)) \\
= 4*n+ 3 & + & (T_{\text{Pr2}}(1) & + & T_{\text{Pr2}}(2) & + & \ldots & + & T_{\text{Pr2}}(n)) \\
= 4*n+ 3 & + & (4+4 & + & T_{\text{Pr2}}(3) & + & \ldots & + & T_{\text{Pr2}}(n)) \\
=4*n+ 11 & + & ((5+3*\sqrt{3}) & + & (5+3*\sqrt{4}) & + & \ldots & + & (5+3*\sqrt{n}))
\end{array}
$$

Nombre de nombres premiers entre 1 et n

Calcul final

Le nombre d'instructions nécessaires à l'exécution de la fonction "nombre de nombre premiers entre 1 et n" est de :

$T_{\text{nprem}}(n)$

$= 3$	$+$	$((4 + T_{Pr2}(1))$	$+$	$(4 + T_{Pr2}(2))$	$+$	$\ldots$	$+$	$(4 + T_{Pr2}(n))$
$= 4*n+ 3$	$+$	$(T_{Pr2}(1)$	$+$	$T_{Pr2}(2)$	$+$	$\ldots$	$+$	$T_{Pr2}(n))$
$= 4*n+ 3$	$+$	$(4+4$	$+$	$T_{Pr2}(3)$	$+$	$\ldots$	$+$	$T_{Pr2}(n))$
$=4*n+ 11$	$+$	$((5+3*\sqrt{3})$	$+$	$(5+3*\sqrt{4})$	$+$	$\ldots$	$+$	$(5+3*\sqrt{n}))$
$=4*n+ 11$	$+$	$5*(n-2)+3*(\sqrt{3}$	$+$	$\sqrt{4}$	$+$	$\ldots$	$+$	$\sqrt{n})$

Nombre de nombres premiers entre 1 et n

Calcul final

Le nombre d'instructions nécessaires à l'exécution de la fonction "nombre de nombre premiers entre 1 et n" est de :

$T_{\text{nprem}}(n)$

$= 3$	$+$	$((4 + T_{Pr2}(1))$	$+$	$(4 + T_{Pr2}(2))$	$+$	$\ldots$	$+$	$(4 + T_{Pr2}(n))$
$= 4*n+ 3$	$+$	$(T_{Pr2}(1)$	$+$	$T_{Pr2}(2)$	$+$	$\ldots$	$+$	$T_{Pr2}(n))$
$= 4*n+ 3$	$+$	$(4+4$	$+$	$T_{Pr2}(3)$	$+$	$\ldots$	$+$	$T_{Pr2}(n))$
$=4*n+ 11$	$+$	$((5+3*\sqrt{3})$	$+$	$(5+3*\sqrt{4})$	$+$	$\ldots$	$+$	$(5+3*\sqrt{n}))$
$=4*n+ 11$	$+$	$5*(n-2)+3*(\sqrt{3}$	$+$	$\sqrt{4}$	$+$	$\ldots$	$+$	$\sqrt{n})$
$=9*n+ 1$	$+$	$3*(\sqrt{3}$	$+$	$\sqrt{4}$	$+$	$\ldots$	$+$	$\sqrt{n})$

Nombre de nombres premiers entre 1 et n : complexité

Calcul final

Le nombre d'instructions nécessaires à l'exécution de la fonction "nombre de nombre premiers entre 1 et n" est donc :

$$T_{\text{nprem}}(n) = 9.n + 1 + 3.(\sqrt{3} + \sqrt{4} + ... + \sqrt{n})$$

Supposons que l'on ne connaît pas de formules qui résume l'expression :

$$(\sqrt{3} + \sqrt{4} + ... + \sqrt{n}).$$

Nombre de nombres premiers entre 1 et n : complexité

Essayons d'encadrer l'expression $(\sqrt{3} + \sqrt{4} + ... + \sqrt{n})$.

- Comme $\forall i = 3, .., n, \sqrt{i} \leq \sqrt{n}$, nous avons :

$$(\sqrt{3} + \sqrt{4} + ... + \sqrt{n}) \leq (n-2).\sqrt{n}.$$

De ce fait :

$$T_{\text{nprem}}(n) \leq 9.n + 1 + 3.(n-2).\sqrt{n},$$

et :

$$T_{\text{nprem}}(n) \in O(\sqrt{n}.n).$$

Nombre de nombres premiers entre 1 et n : complexité

- De même, $\forall i = 3, .., n, \sqrt{i} \geq 1$, nous avons :

$$(\sqrt{3} + \sqrt{4} + ... + \sqrt{n}) \geq (n - 2).$$

De ce fait :

$$T_{\text{nprem}}(n) \geq 9.n + 1 + 3.(n - 2),$$

et

$$T_{\text{nprem}}(n) \in \Omega(n).$$

- En conclusion, la complexité temporelle de la fonction "nombre de nombre premiers entre 1 et n" est comprise entre $c_1 n$ et $c_2 n.\sqrt{n}$ où c_1 etc_2 sont des constantes.

Favorables, défavorables, moyens

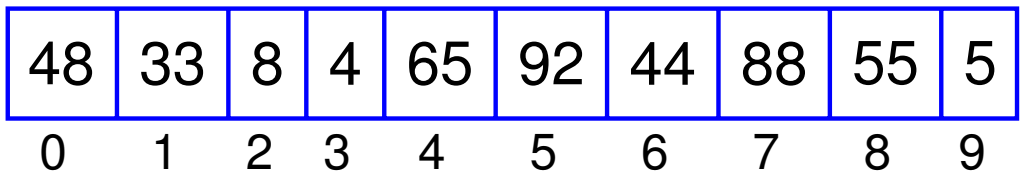
Exemple : Tri par bulles

Supposons que nous devons trier des tableaux d'entiers de n éléments par bulles.

- **1.** L'idée est de parcourir le tableau et à chaque fois qu'il y a deux éléments successifs non-ordonnés on les permute.
- **2.** L'algorithme s'arrête si à un parcours du tableau aucune permutation n'a été effectué (sinon on répète l'étape **1**).

Exemple : tri par bulles

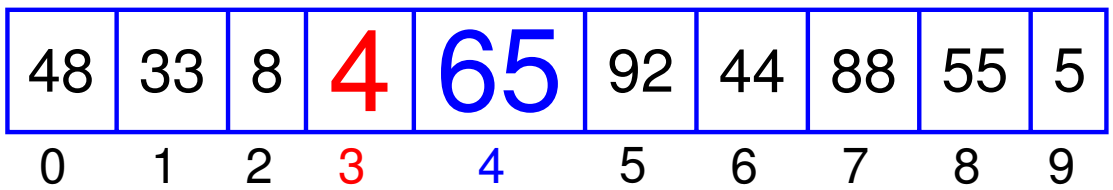
Voici le tableau initial à trier :



Exemple : tri par bulles

Traitement de l'iteration 1 :

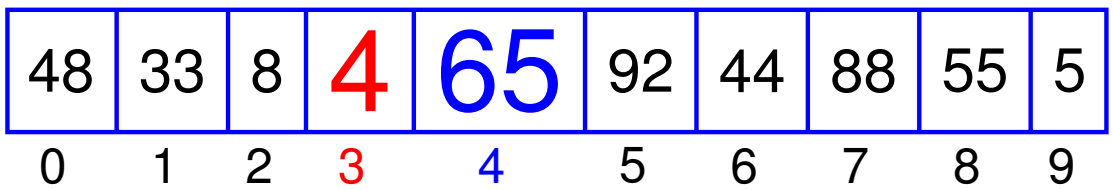
Les deux cases sucessives Tab[3]=4 et Tab[4]=65 ne sont pas ordonnées.



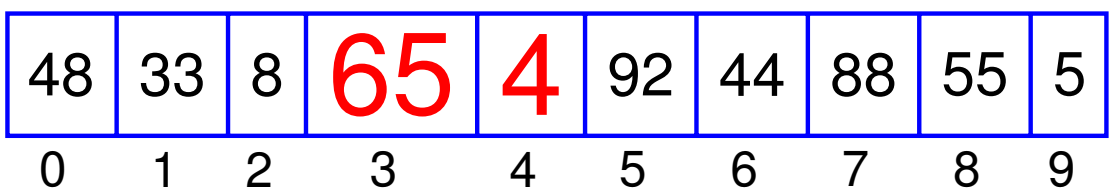
Exemple : tri par bulles

Traitement de l'iteration 1 :

Les deux cases sucessives Tab[3]=4 et Tab[4]=65 ne sont pas ordonnées.



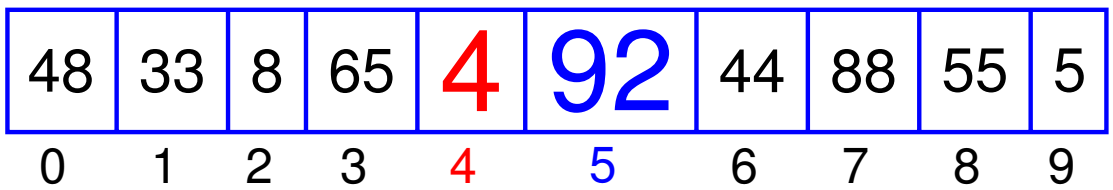
L'échange entre Tab[3]=4 et Tab[4]=65 donne :



Exemple : tri par bulles

Traitement de l'iteration 1 :

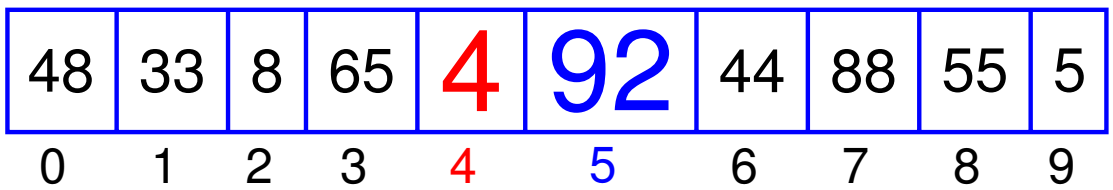
Les deux cases sucessives Tab[4]=4 et Tab[5]=92 ne sont pas ordonnées.



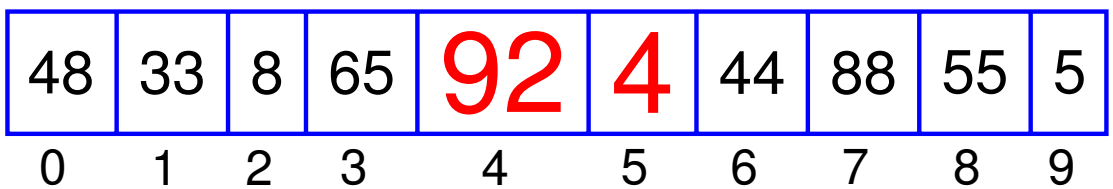
Exemple : tri par bulles

Traitement de l'iteration 1 :

Les deux cases sucessives Tab[4]=4 et Tab[5]=92 ne sont pas ordonnées.



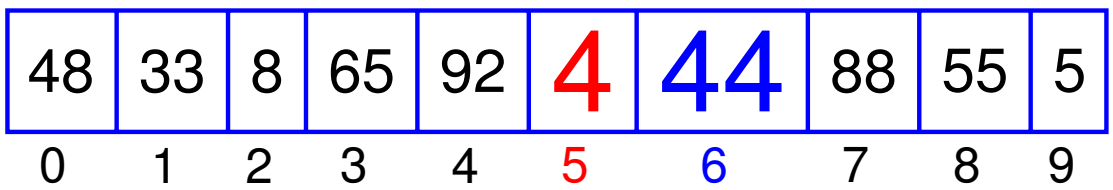
L'échange entre Tab[4]=4 et Tab[5]=92 donne :



Exemple : tri par bulles

Traitement de l'iteration 1 :

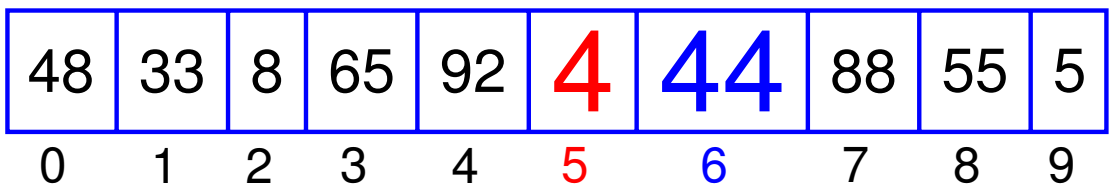
Les deux cases sucessives Tab[5]=4 et Tab[6]=44 ne sont pas ordonnées.



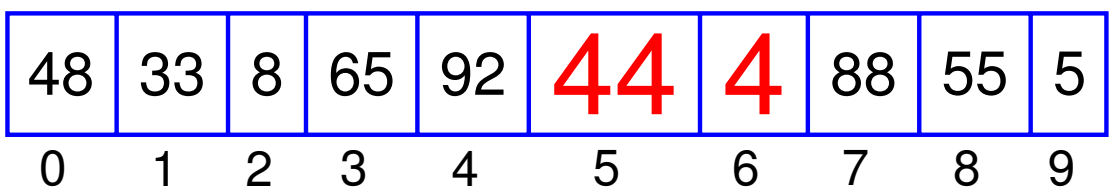
Exemple : tri par bulles

Traitement de l'iteration 1 :

Les deux cases sucessives Tab[5]=4 et Tab[6]=44 ne sont pas ordonnées.



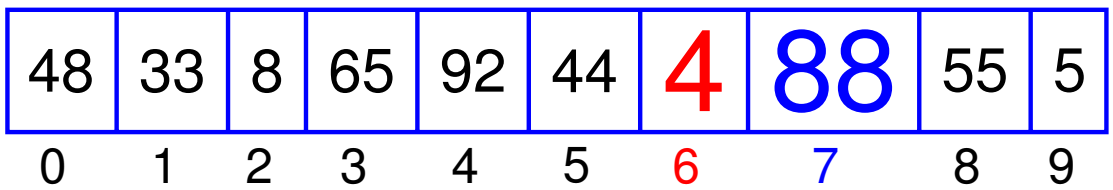
L'échange entre Tab[5]=4 et Tab[6]=44 donne :



Exemple : tri par bulles

Traitement de l'iteration 1 :

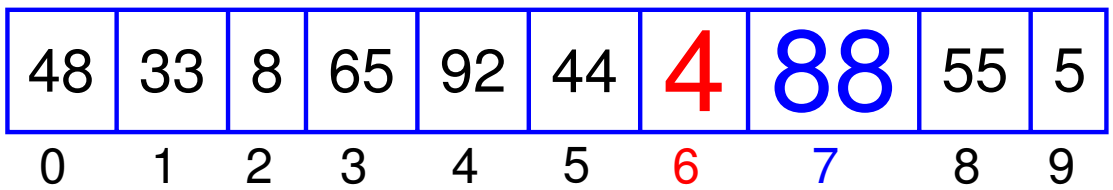
Les deux cases sucessives Tab[6]=4 et Tab[7]=88 ne sont pas ordonnées.



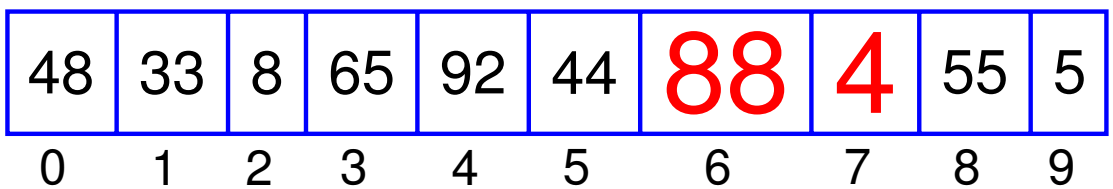
Exemple : tri par bulles

Traitement de l'iteration 1 :

Les deux cases sucessives Tab[6]=4 et Tab[7]=88 ne sont pas ordonnées.



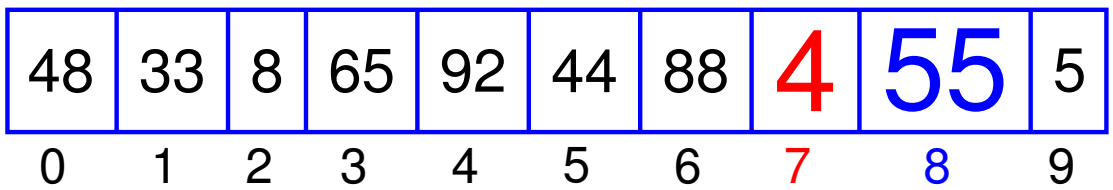
L'échange entre Tab[6]=4 et Tab[7]=88 donne :



Exemple : tri par bulles

Traitement de l'iteration 1 :

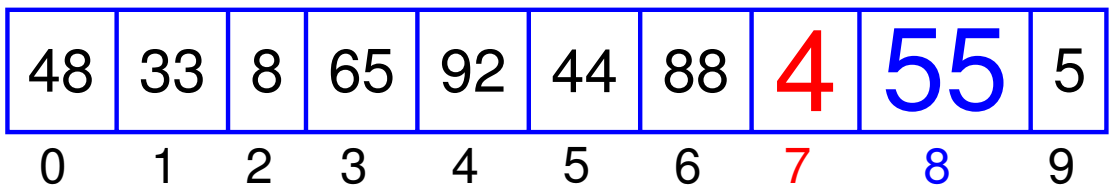
Les deux cases sucessives Tab[7]=4 et Tab[8]=55 ne sont pas ordonnées.



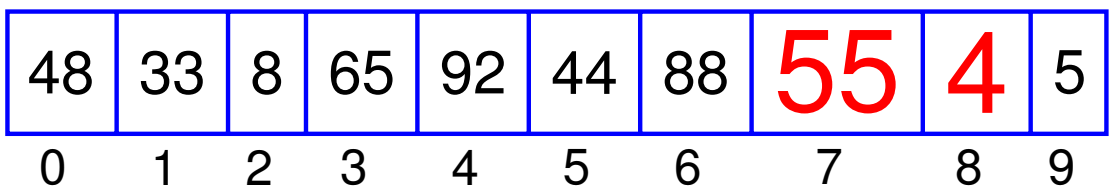
Exemple : tri par bulles

Traitement de l'iteration 1 :

Les deux cases sucessives Tab[7]=4 et Tab[8]=55 ne sont pas ordonnées.



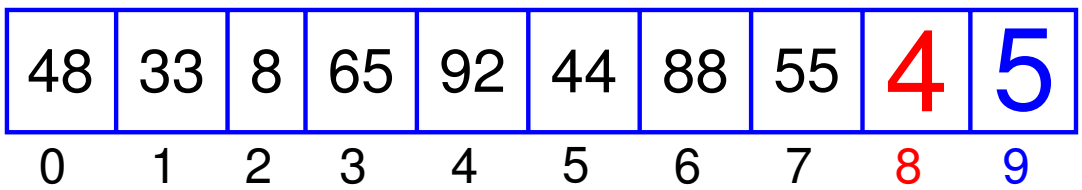
L'échange entre Tab[7]=4 et Tab[8]=55 donne :



Exemple : tri par bulles

Traitement de l'iteration 1 :

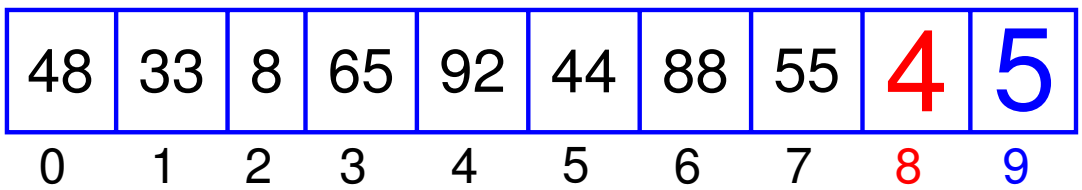
Les deux cases sucessives Tab[8]=4 et Tab[9]=5 ne sont pas ordonnées.



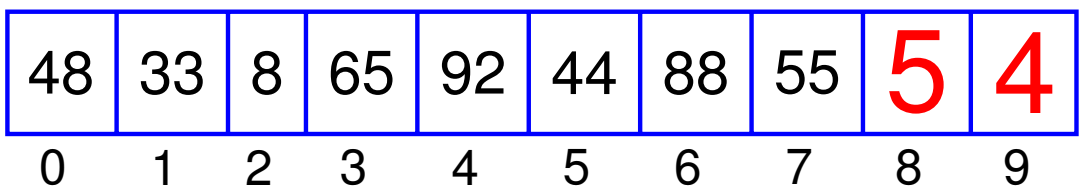
Exemple : tri par bulles

Traitement de l'iteration 1 :

Les deux cases sucessives Tab[8]=4 et Tab[9]=5 ne sont pas ordonnées.



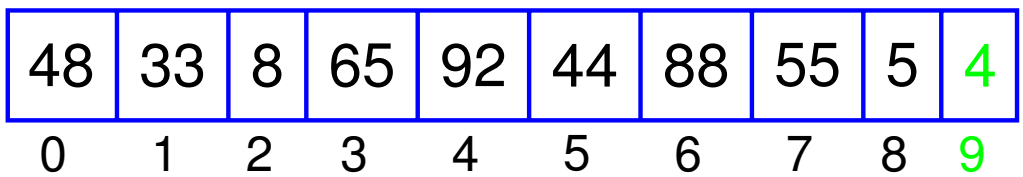
L'échange entre Tab[8]=4 et Tab[9]=5 donne :



Exemple : tri par bulles

Fin de l'itération 1 :

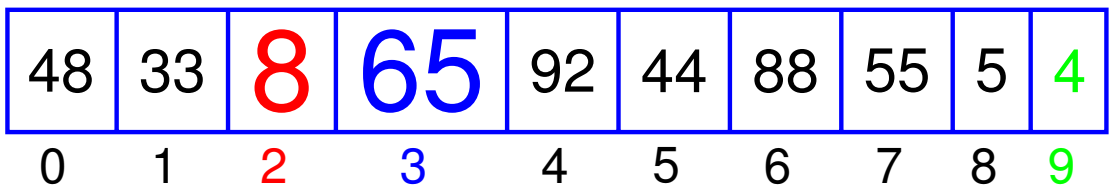
Après l'itération 1, l'élément Tab[9]=4 est à sa bonne place



Exemple : tri par bulles

Traitement de l'iteration 2 :

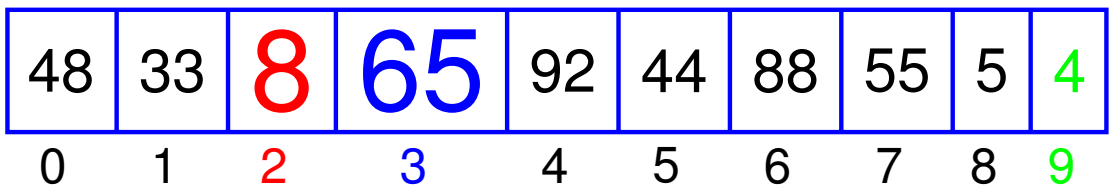
Les deux cases sucessives Tab[2]=8 et Tab[3]=65 ne sont pas ordonnées.



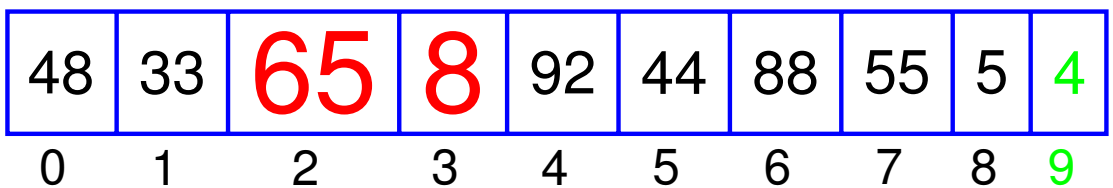
Exemple : tri par bulles

Traitement de l'iteration 2 :

Les deux cases sucessives Tab[2]=8 et Tab[3]=65 ne sont pas ordonnées.



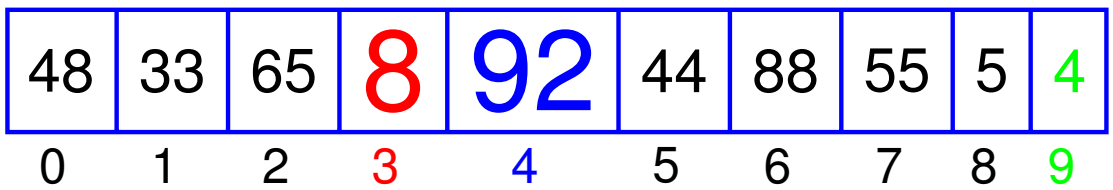
L'échange entre Tab[2]=8 et Tab[3]=65 donne :



Exemple : tri par bulles

Traitement de l'iteration 2 :

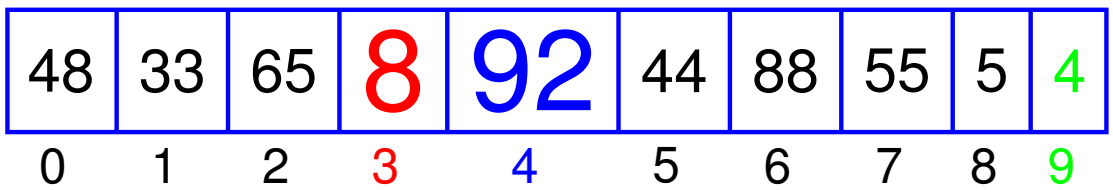
Les deux cases sucessives Tab[3]=8 et Tab[4]=92 ne sont pas ordonnées.



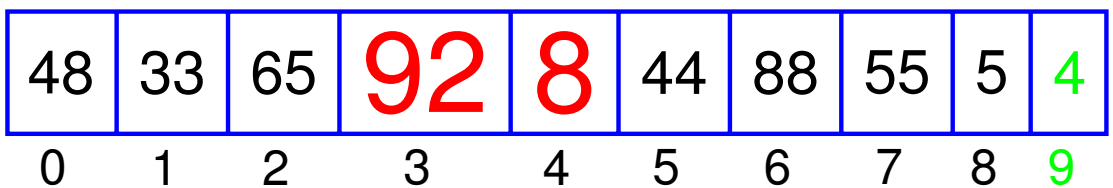
Exemple : tri par bulles

Traitement de l'iteration 2 :

Les deux cases sucessives Tab[3]=8 et Tab[4]=92 ne sont pas ordonnées.



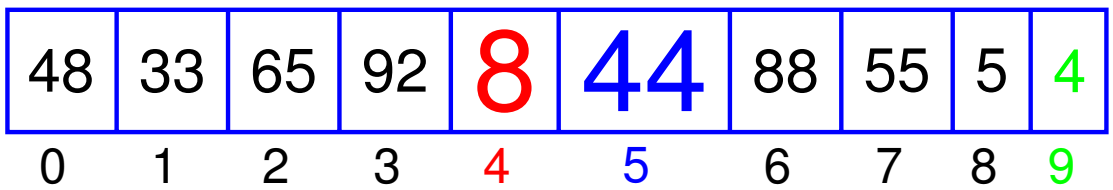
L'échange entre Tab[3]=8 et Tab[4]=92 donne :



Exemple : tri par bulles

Traitement de l'iteration 2 :

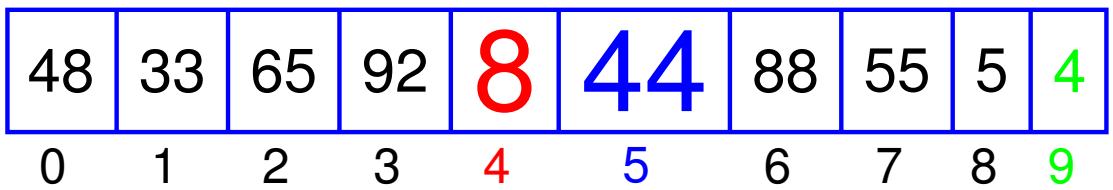
Les deux cases sucessives Tab[4]=8 et Tab[5]=44 ne sont pas ordonnées.



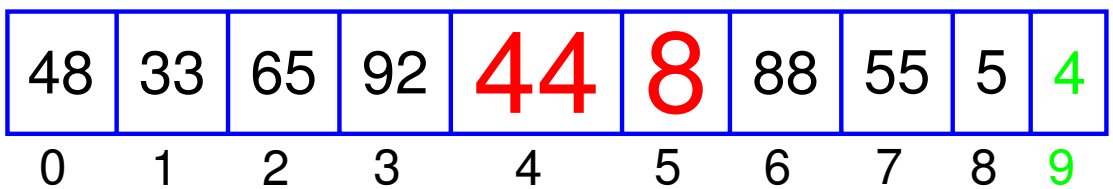
Exemple : tri par bulles

Traitement de l'iteration 2 :

Les deux cases sucessives Tab[4]=8 et Tab[5]=44 ne sont pas ordonnées.



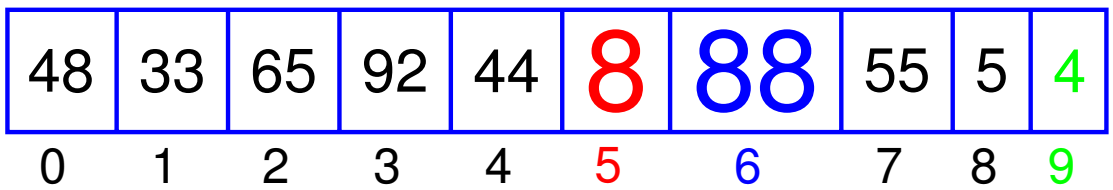
L'échange entre Tab[4]=8 et Tab[5]=44 donne :



Exemple : tri par bulles

Traitement de l'iteration 2 :

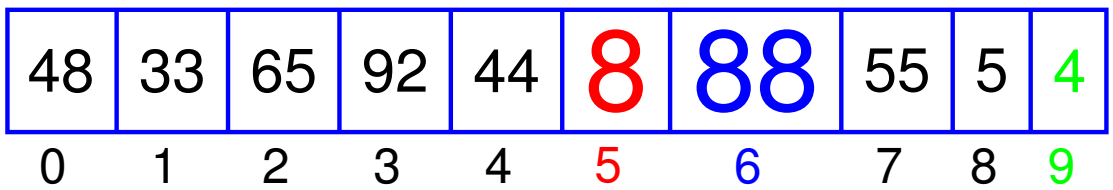
Les deux cases sucessives Tab[5]=8 et Tab[6]=88 ne sont pas ordonnées.



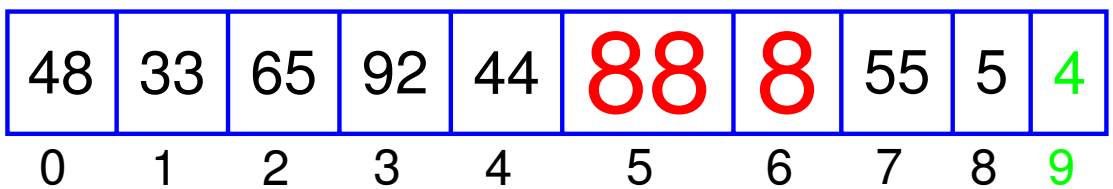
Exemple : tri par bulles

Traitement de l'iteration 2 :

Les deux cases sucessives Tab[5]=8 et Tab[6]=88 ne sont pas ordonnées.



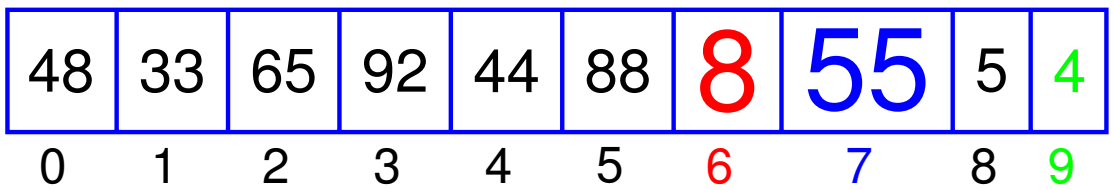
L'échange entre Tab[5]=8 et Tab[6]=88 donne :



Exemple : tri par bulles

Traitement de l'iteration 2 :

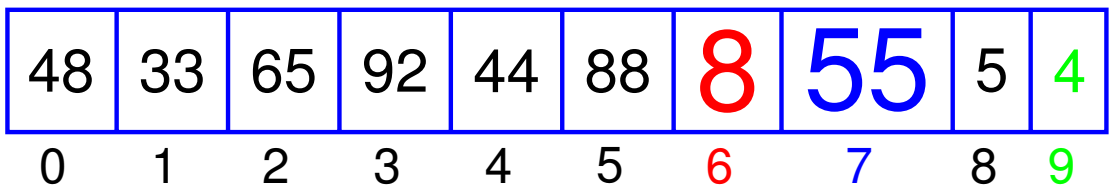
Les deux cases sucessives Tab[6]=8 et Tab[7]=55 ne sont pas ordonnées.



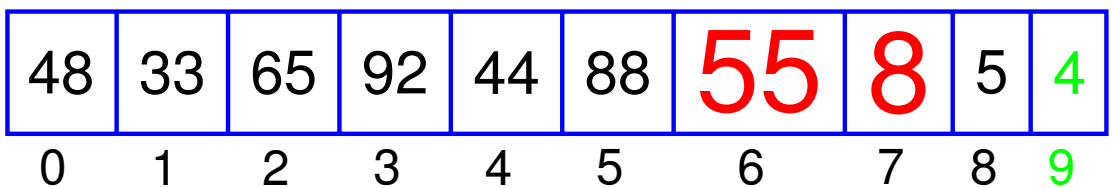
Exemple : tri par bulles

Traitement de l'iteration 2 :

Les deux cases sucessives Tab[6]=8 et Tab[7]=55 ne sont pas ordonnées.



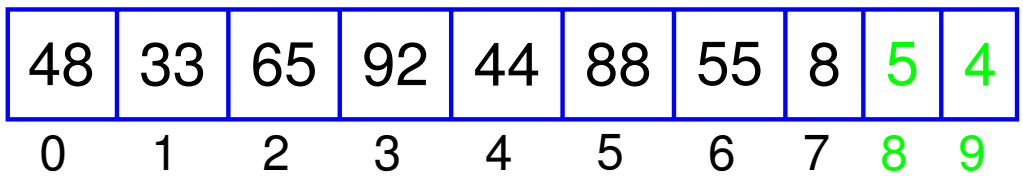
L'échange entre Tab[6]=8 et Tab[7]=55 donne :



Exemple : tri par bulles

Fin de l'itération 2 :

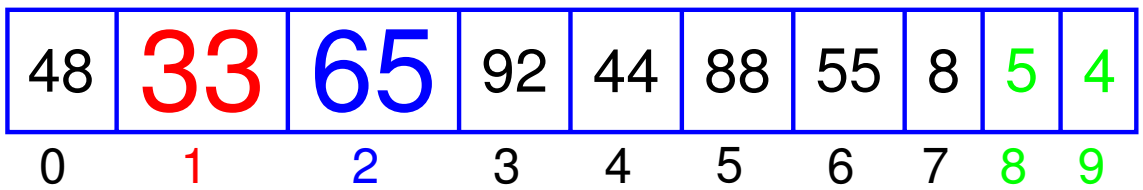
Après l'itération 2, l'élément Tab[8]=5 est à sa bonne place



Exemple : tri par bulles

Traitement de l'iteration 3 :

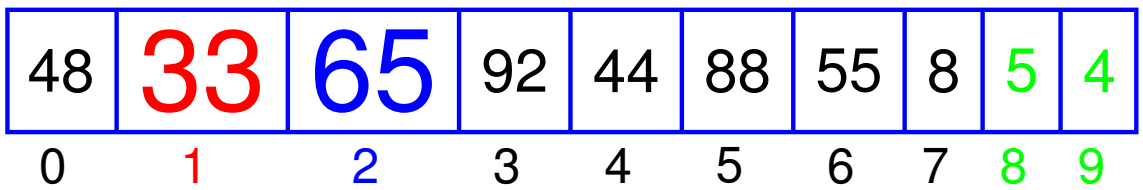
Les deux cases sucessives Tab[1]=33 et Tab[2]=65 ne sont pas ordonnées.



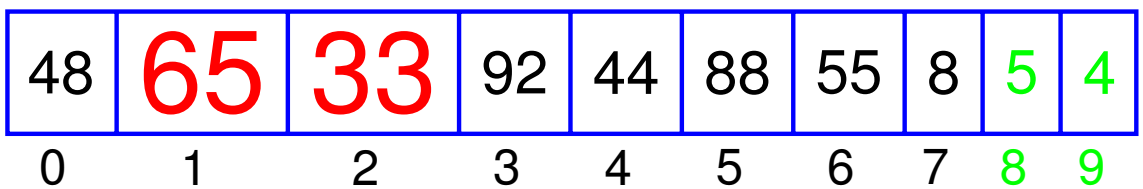
Exemple : tri par bulles

Traitement de l'iteration 3 :

Les deux cases sucessives Tab[1]=33 et Tab[2]=65 ne sont pas ordonnées.



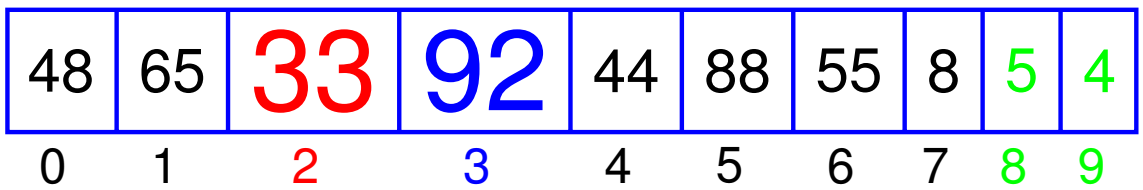
L'échange entre Tab[1]=33 et Tab[2]=65 donne :



Exemple : tri par bulles

Traitement de l'iteration 3 :

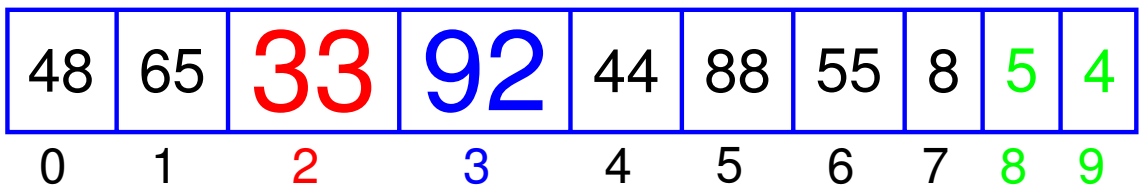
Les deux cases sucessives Tab[2]=33 et Tab[3]=92 ne sont pas ordonnées.



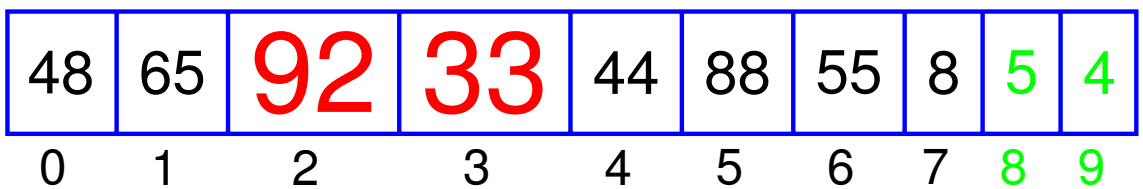
Exemple : tri par bulles

Traitement de l'iteration 3 :

Les deux cases sucessives Tab[2]=33 et Tab[3]=92 ne sont pas ordonnées.



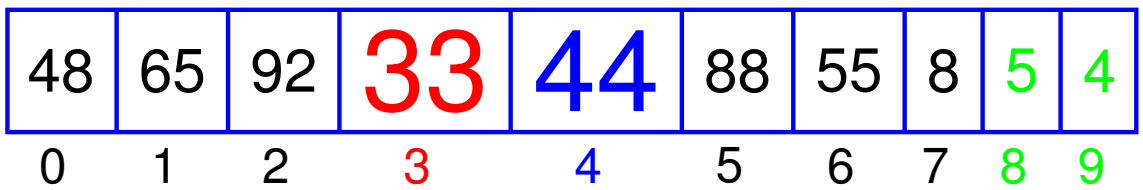
L'échange entre Tab[2]=33 et Tab[3]=92 donne :



Exemple : tri par bulles

Traitement de l'iteration 3 :

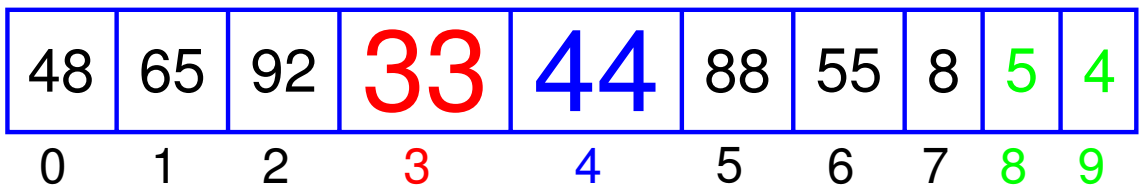
Les deux cases sucessives Tab[3]=33 et Tab[4]=44 ne sont pas ordonnées.



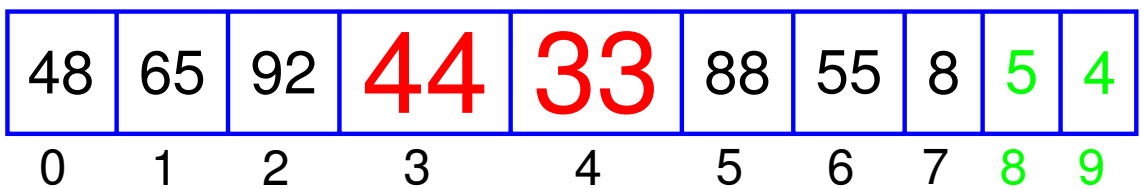
Exemple : tri par bulles

Traitement de l'iteration 3 :

Les deux cases sucessives Tab[3]=33 et Tab[4]=44 ne sont pas ordonnées.



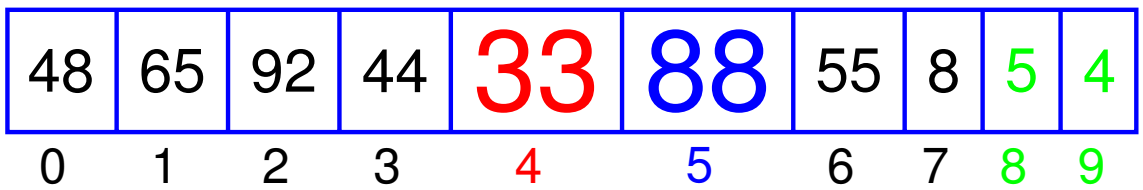
L'échange entre Tab[3]=33 et Tab[4]=44 donne :



Exemple : tri par bulles

Traitement de l'iteration 3 :

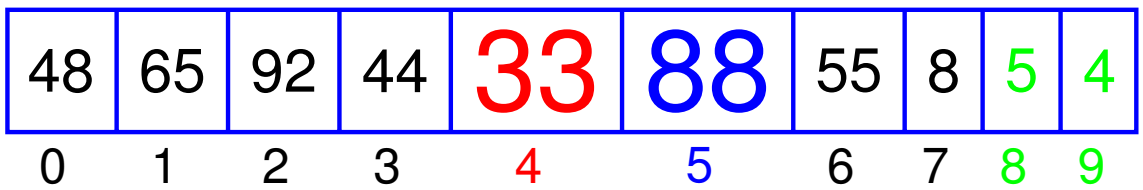
Les deux cases sucessives Tab[4]=33 et Tab[5]=88 ne sont pas ordonnées.



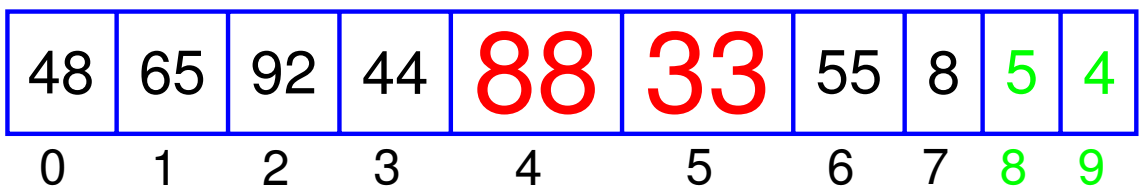
Exemple : tri par bulles

Traitement de l'iteration 3 :

Les deux cases sucessives Tab[4]=33 et Tab[5]=88 ne sont pas ordonnées.



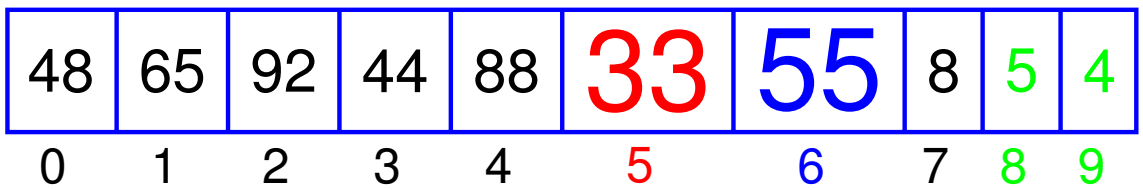
L'échange entre Tab[4]=33 et Tab[5]=88 donne :



Exemple : tri par bulles

Traitement de l'iteration 3 :

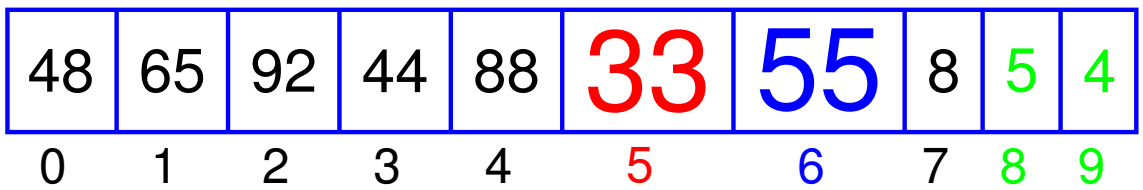
Les deux cases sucessives Tab[5]=33 et Tab[6]=55 ne sont pas ordonnées.



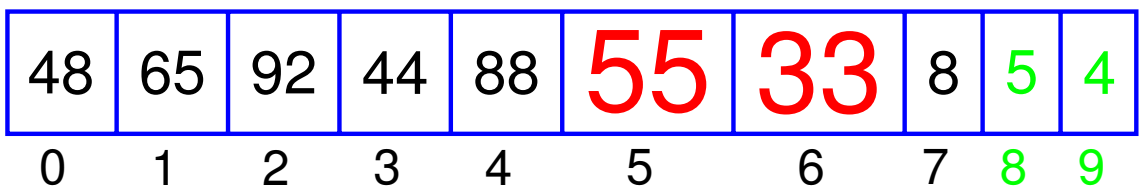
Exemple : tri par bulles

Traitement de l'iteration 3 :

Les deux cases sucessives Tab[5]=33 et Tab[6]=55 ne sont pas ordonnées.



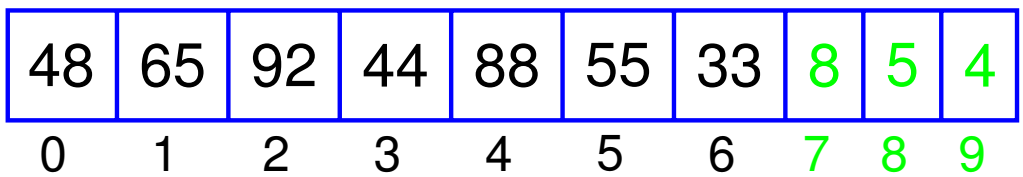
L'échange entre Tab[5]=33 et Tab[6]=55 donne :



Exemple : tri par bulles

Fin de l'itération 3 :

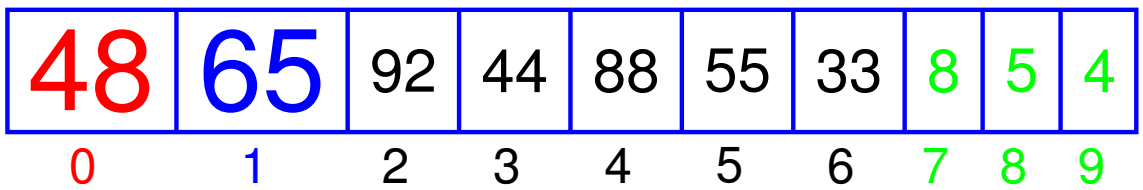
Après l'itération 3, l'élément Tab[7]=8 est à sa bonne place



Exemple : tri par bulles

Traitement de l'iteration 4 :

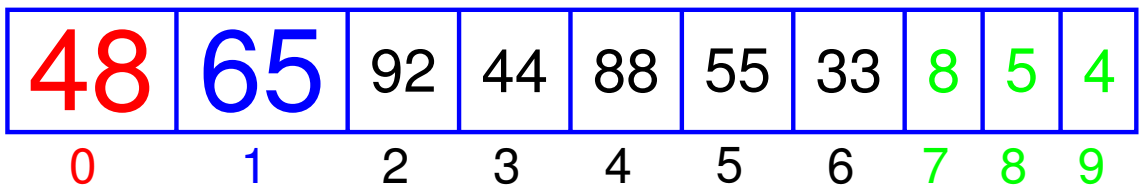
Les deux cases sucessives Tab[0]=48 et Tab[1]=65 ne sont pas ordonnées.



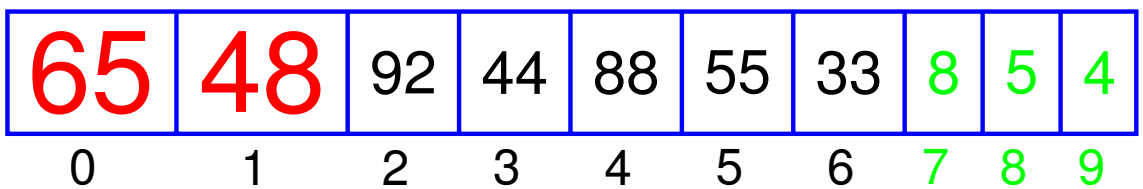
Exemple : tri par bulles

Traitement de l'iteration 4 :

Les deux cases sucessives Tab[0]=48 et Tab[1]=65 ne sont pas ordonnées.



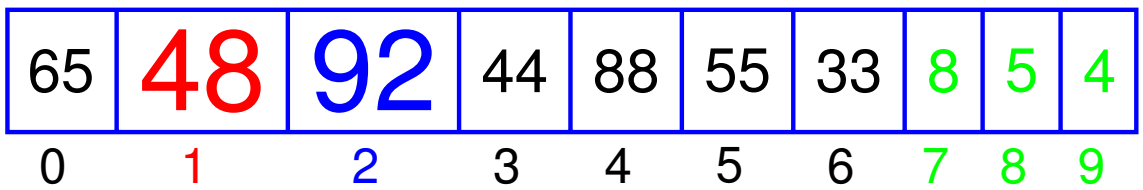
L'échange entre Tab[0]=48 et Tab[1]=65 donne :



Exemple : tri par bulles

Traitement de l'iteration 4 :

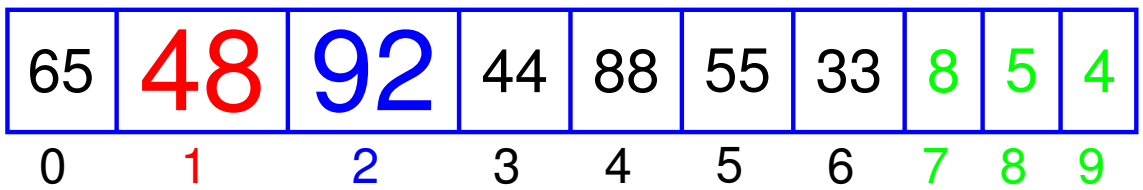
Les deux cases sucessives Tab[1]=48 et Tab[2]=92 ne sont pas ordonnées.



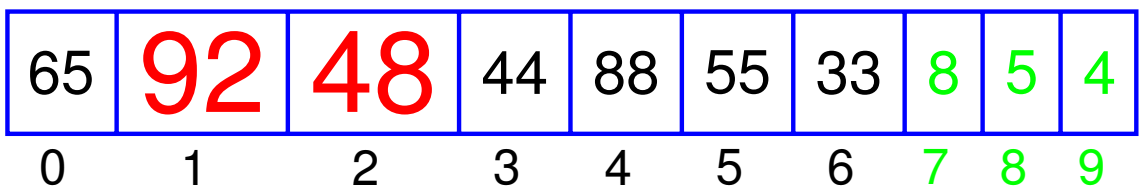
Exemple : tri par bulles

Traitement de l'iteration 4 :

Les deux cases sucessives Tab[1]=48 et Tab[2]=92 ne sont pas ordonnées.



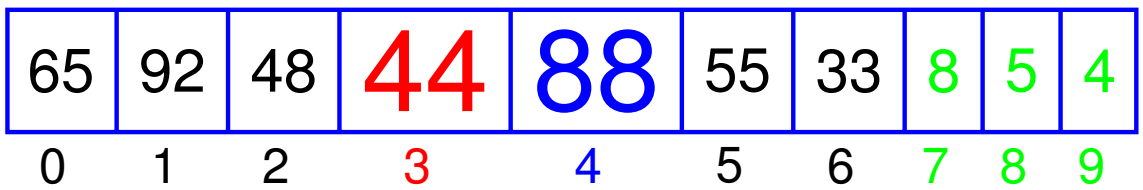
L'échange entre Tab[1]=48 et Tab[2]=92 donne :



Exemple : tri par bulles

Traitement de l'iteration 4 :

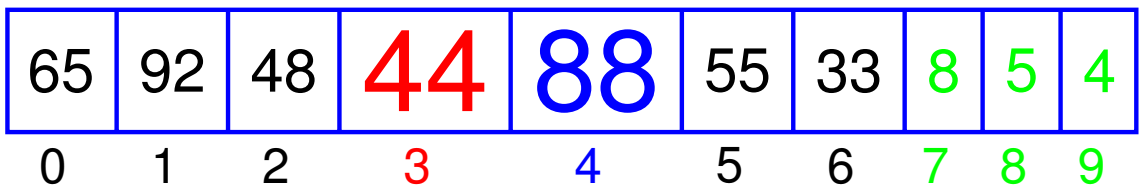
Les deux cases sucessives Tab[3]=44 et Tab[4]=88 ne sont pas ordonnées.



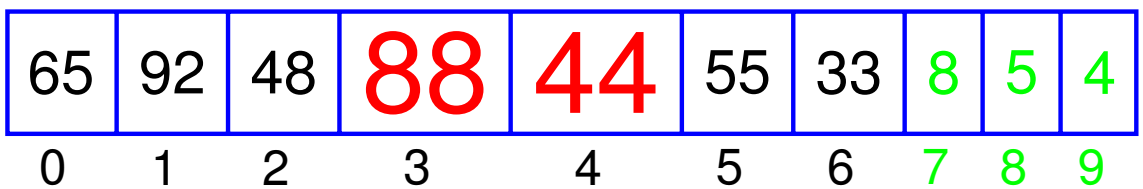
Exemple : tri par bulles

Traitement de l'iteration 4 :

Les deux cases sucessives Tab[3]=44 et Tab[4]=88 ne sont pas ordonnées.



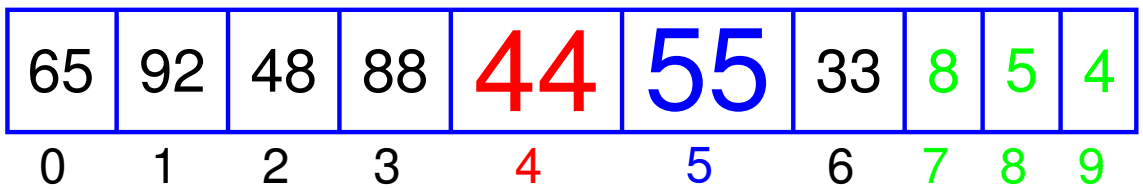
L'échange entre Tab[3]=44 et Tab[4]=88 donne :



Exemple : tri par bulles

Traitement de l'iteration 4 :

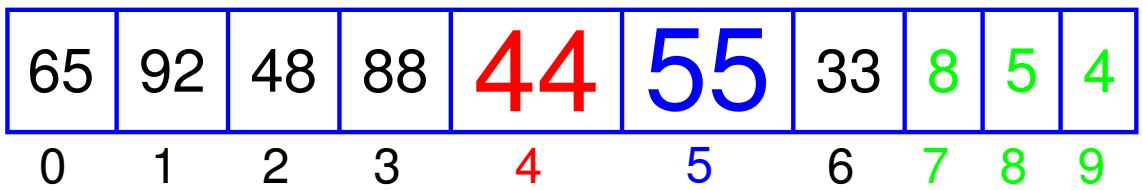
Les deux cases sucessives Tab[4]=44 et Tab[5]=55 ne sont pas ordonnées.



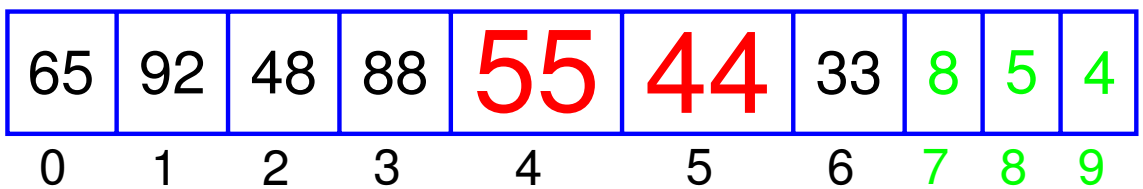
Exemple : tri par bulles

Traitement de l'iteration 4 :

Les deux cases sucessives Tab[4]=44 et Tab[5]=55 ne sont pas ordonnées.



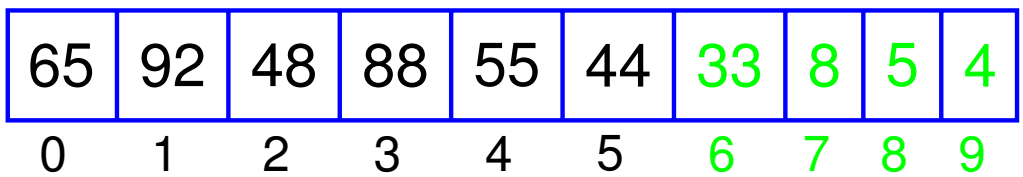
L'échange entre Tab[4]=44 et Tab[5]=55 donne :



Exemple : tri par bulles

Fin de l'itération 4 :

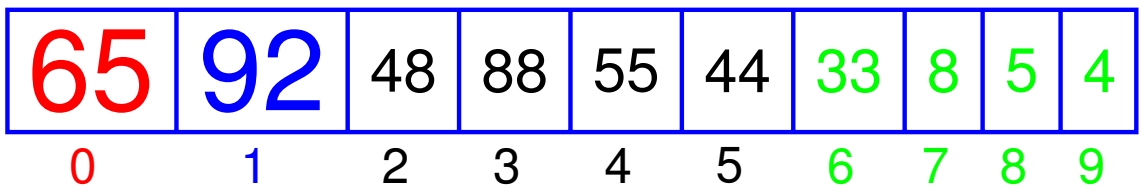
Après l'itération 4, l'élément Tab[6]=33 est à sa bonne place



Exemple : tri par bulles

Traitement de l'iteration 5 :

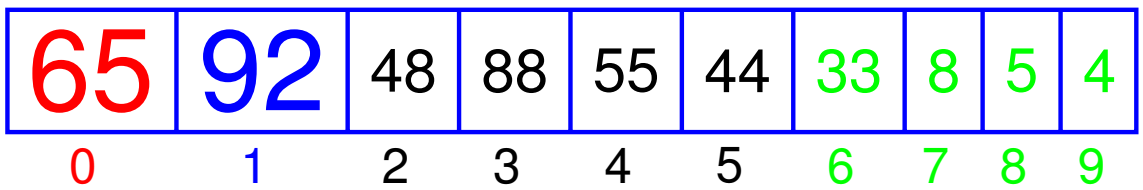
Les deux cases sucessives Tab[0]=65 et Tab[1]=92 ne sont pas ordonnées.



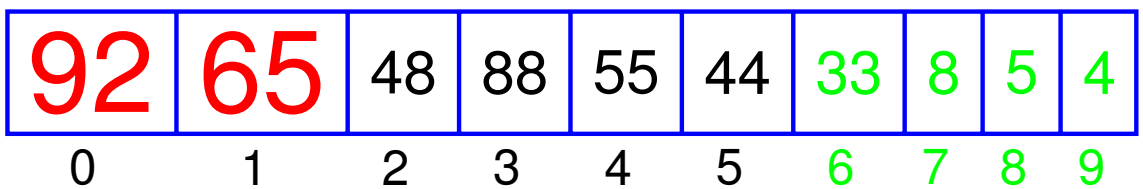
Exemple : tri par bulles

Traitement de l'iteration 5 :

Les deux cases sucessives Tab[0]=65 et Tab[1]=92 ne sont pas ordonnées.



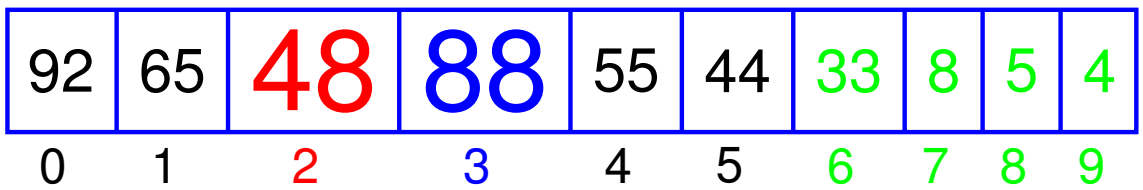
L'échange entre Tab[0]=65 et Tab[1]=92 donne :



Exemple : tri par bulles

Traitement de l'iteration 5 :

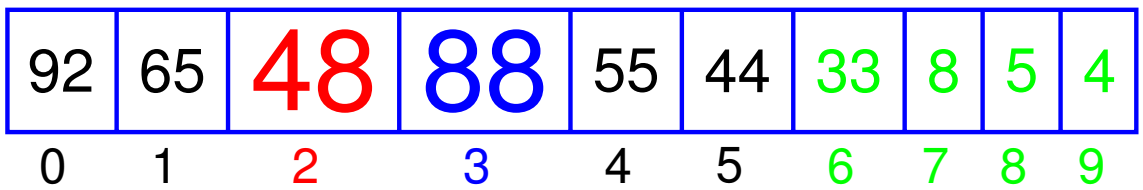
Les deux cases sucessives Tab[2]=48 et Tab[3]=88 ne sont pas ordonnées.



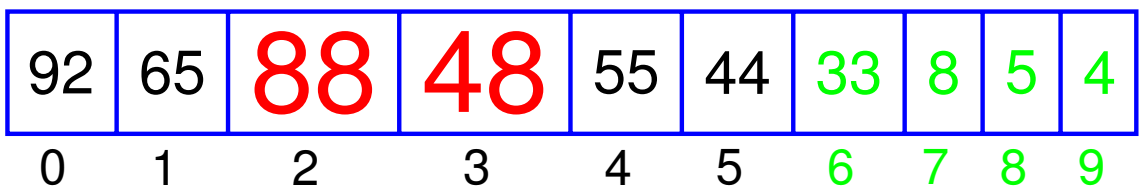
Exemple : tri par bulles

Traitement de l'iteration 5 :

Les deux cases sucessives Tab[2]=48 et Tab[3]=88 ne sont pas ordonnées.



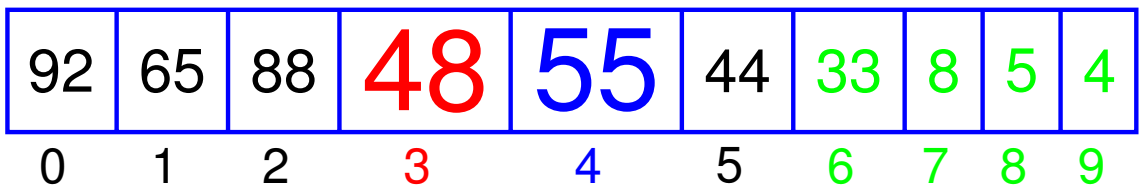
L'échange entre Tab[2]=48 et Tab[3]=88 donne :



Exemple : tri par bulles

Traitement de l'iteration 5 :

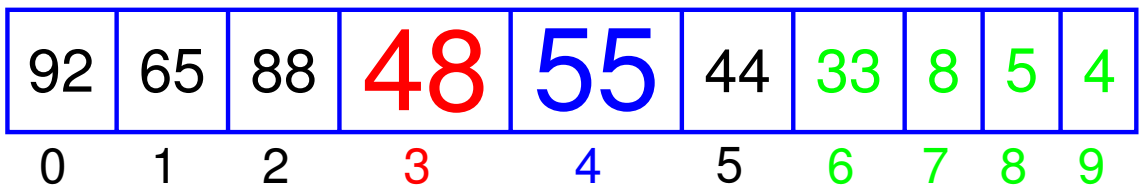
Les deux cases sucessives Tab[3]=48 et Tab[4]=55 ne sont pas ordonnées.



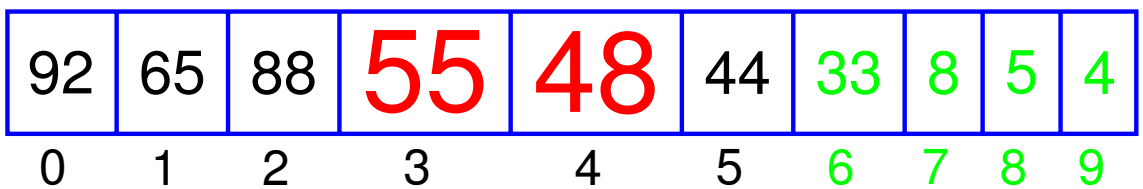
Exemple : tri par bulles

Traitement de l'iteration 5 :

Les deux cases sucessives Tab[3]=48 et Tab[4]=55 ne sont pas ordonnées.



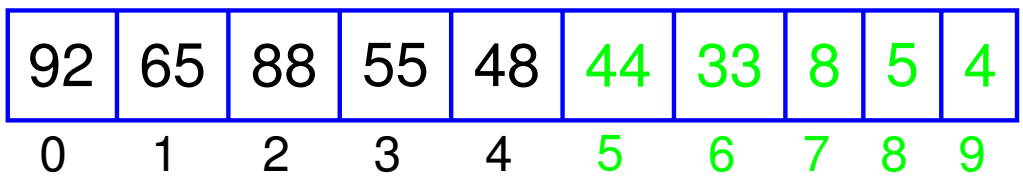
L'échange entre Tab[3]=48 et Tab[4]=55 donne :



Exemple : tri par bulles

Fin de l'itération 5 :

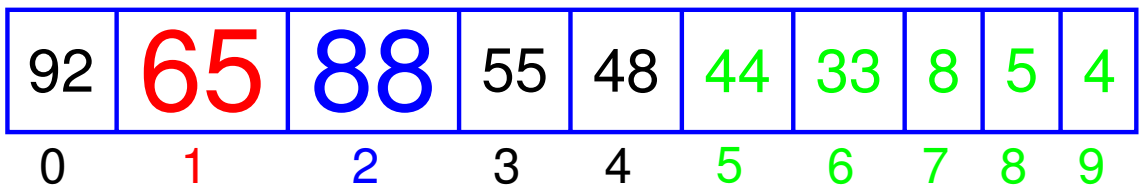
Après l'itération 5, l'élément Tab[5]=44 est à sa bonne place



Exemple : tri par bulles

Traitement de l'iteration 6 :

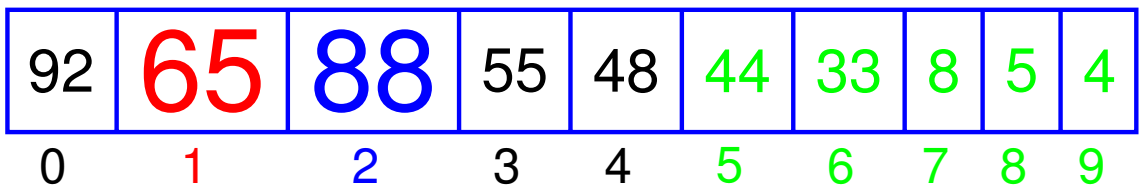
Les deux cases sucessives Tab[1]=65 et Tab[2]=88 ne sont pas ordonnées.



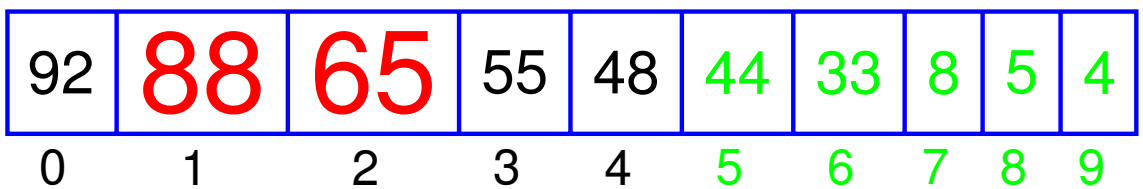
Exemple : tri par bulles

Traitement de l'iteration 6 :

Les deux cases sucessives Tab[1]=65 et Tab[2]=88 ne sont pas ordonnées.



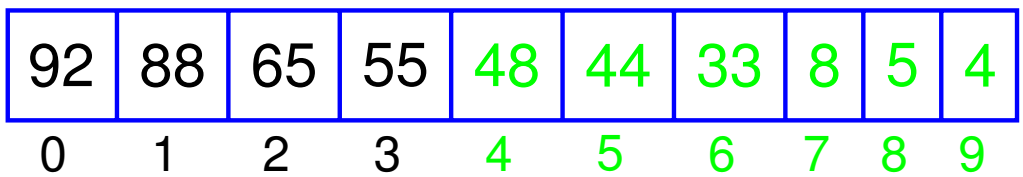
L'échange entre Tab[1]=65 et Tab[2]=88 donne :



Exemple : tri par bulles

Fin de l'itération 6 :

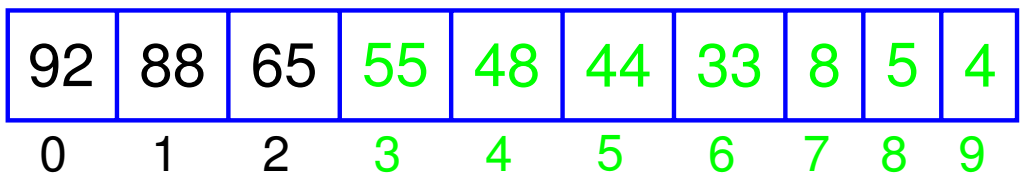
Après l'itération 6, l'élément Tab[4]=48 est à sa bonne place



Exemple : tri par bulles

Fin de l'itération 7 :

Après l'itération 7, l'élément Tab[3]=55 est à sa bonne place



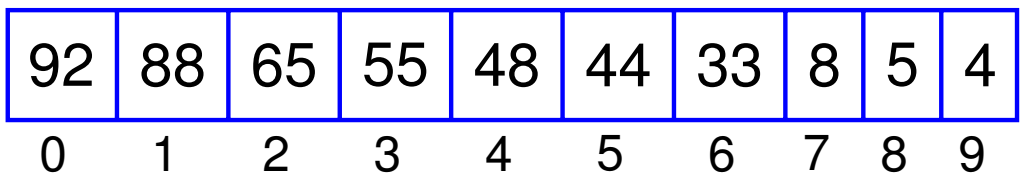
Exemple : tri par bulles

Fin du déroulement :

L'application du tri par bulles a nécessité :

- 24 échanges
- 7 iterations

Après l'application de l'algorithme on obtient :



Favorables, défavorables, moyens

Principe

- Produire tous les jeux de données possibles A.
  - Par exemple, pour analyser un algorithme de tri, pour chaque taille possible, on génère tous les tableaux possibles.
- Pour chaque instance a (jeu de données) de A on calcule le temps d'exécution $T(a)$ (où le nombre d'instructions élémentaires nécessaires à l'exécution de l'algorithme sur l'instance a) .

Favorables, défavorables, moyens

Définitions

- **Favorable** Le cas favorable est défini par :

$$t_{\min} = min\{T(a) : a \in A\}.$$

- **Défavorable** Le cas défavorable est défini par :

$$t_{\max} = max\{T(a) : a \in A\}.$$

- **Moyen** Le cas moyen (avec une distribution uniforme sur les instances) est défini par :

$$t_{\text{moyen}} = \frac{\sum_{a \in A} T(a)}{|A|}.$$

Retour sur le cas moyen

- En général, les jeux de données n'ont pas la même probabilité d'être traitée par un algorithme
- Par exemple, supposons que l'on trie les notes des étudiants obtenues à la première session de L3. Alors :
  - L'instance où toutes les notes sont égales à "0" est impossible.
  - L'instance où toutes les notes sont égales à "20" est peu probable.
  - Les instances où la moyenne des notes est "12" (avec un écart type de 1.3) sont les plus typiques (vraissemblables).

Retour sur le cas moyen

- De ce fait, chaque instance a lui est associé un degré de probabilité $P(a)$ que l'algorithme traitera cette instance.
- La complexité en moyenne dans ce cas est de :

$$t_{\text{moyen}} = \sum_{a \in A} T(a).P(a).$$

- En pratique il est difficile d'exhiber une telle probabilité, c'est la raison pour laquelle une distribution de probabilité uniforme est souvent supposée pour le calcul de la complexité en moyenne.