Master Informatique - M1 - UE Complexité
Chapitre 2 : Rappels, algorithmique et complexité

Philippe Jégou

Laboratoire d'Informatique et Systèmes - LIS - UMR CNRS 7020

Équipe COALA - COntraintes, ALgorithmes et Applications

(Algorithmique et Complexité de l'Intelligence Artificielle)

Campus de Saint-Jérôme

Département Informatique et Interactions

Faculté des Sciences

Université d'Aix-Marseille

philippe.jegou@univ-amu.fr

4 septembre 2022

Plan

1. Analyse de la Complexité des Algorithmes
    - Motivations
    - Un modèle d'analyse fondé sur quelques principes
    - Quelques exemples moins triviaux (mais du niveau L2...)
    - Notations Θ, O, et Ω : définitions formelles
    - Incidence du codage et précautions pour le modèle d'analyse
    - De la théorie (complexité) à la pratique (efficacité pratique)
2. Graphes et Algorithmes : des rappels
    - Notions de base
        - Présentation
        - Définitions et terminologie
        - Représentation machine
    - Chemins, Parcours et ConnexitéS
        - Chemins, chaînes, parcours et numérotations
        - Connexité et forte connexité
3. Rappels : pour conclure

Plan

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Objectif : savoir si un algorithme puis un programme l'implémentant, seront efficaces en termes de temps d'exécution

⇒ estimer les temps de calcul

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

Objectif : savoir si un algorithme puis un programme l'implémentant, seront efficaces en termes de temps d'exécution

$\Rightarrow$ estimer les temps de calcul

Mais :

- implémenter peut coûter tres cher (hommes/mois)
- il faut estimer le temps avant d'implémenter !

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

Objectif : savoir si un algorithme puis un programme l'implémentant, seront efficaces en termes de temps d'exécution

$\Rightarrow$ estimer les temps de calcul

Mais :

- implémenter peut coûter tres cher (hommes/mois)
- il faut estimer le temps avant d'implémenter !

Remarques :

- algorithme : c'est un objet abstrait donc temps de calcul "théorique"
- programme sur un environnement matériel/systeme donné :
  temps de calcul "physique" (temps au chronomètre)

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ?
  Pas impossible... mais sur quels jeux de données ?

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ? Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
    - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ?
  Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
  - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données
  - tri à bulles : efficacité qui évolue très mal avec la taille de la donnée
    - si 15 minutes sur un tableau d'une certaine taille
    - quel temps pour un tableau contenant 10 fois plus d'élements ?

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ?
  Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
  - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données
  - tri à bulles : efficacité qui évolue très mal avec la taille de la donnée
    - si 15 minutes sur un tableau d'une certaine taille
    - quel temps pour un tableau contenant 10 fois plus d'élements ?

    10 fois plus de temps (150 minutes = 2h30) ?

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ? Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
    - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données
    - tri à bulles : efficacité qui évolue très mal avec la taille de la donnée
      - si 15 minutes sur un tableau d'une certaine taille
      - quel temps pour un tableau contenant 10 fois plus d'élements ?

      10 fois plus de temps (150 minutes = 2h30) ?

      Non : 100 fois plus de temps ! (1500 minutes = 25 heures)

      Le temps de calcul du tri à bulles croît de façon quadratique (en n^2) en fonction de la taille n du tableau à trier (donc pas de façon linéaire)

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver à concevoir les algorithmes les plus rapides possibles avant d'implémenter

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver à concevoir les algorithmes les plus rapides possibles avant d'implémenter
- Donc, s'affranchir des aspects matériels dans un premier temps

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver à concevoir les algorithmes les plus rapides possibles avant d'implémenter
- Donc, s'affranchir des aspects matériels dans un premier temps
- Il faut s'appuyer sur un modéle theorique d'évaluation du temps d'exécution

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver à concevoir les algorithmes les plus rapides possibles avant d'implémenter
- Donc, s'affranchir des aspects matériels dans un premier temps
- Il faut s'appuyer sur un modéle theorique d'évaluation du temps d'exécution

Remarque : on laisse de côté la question du coût en ressource mémoire (complexité en espace) au profit du coût en temps (complexité en temps) crucial dans cette UE.

Plan

Un modèle d'analyse fondé sur quelques principes

Objectif : introduire un modèle d'analyse du temps théorique d'exécution des algorithmes

Ce modèle doit :

- s'affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)

Un modèle d'analyse fondé sur quelques principes

Objectif : introduire un modèle d'analyse du temps théorique d'exécution des algorithmes

Ce modèle doit :

- s'affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)
- être utilisable pour évaluer au mieux les temps d'exécution des programmes
  (programme = algorithme écrit dans un langage de programmation)

Un modèle d'analyse fondé sur quelques principes

Objectif : introduire un modèle d'analyse du temps théorique d'exécution des algorithmes

Ce modèle doit :

- s'affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)
- être utilisable pour évaluer au mieux les temps d'exécution des programmes
  (programme = algorithme écrit dans un langage de programmation)
- envisager tous les cas de figures possibles pour les données en entrée

Un modèle d'analyse fondé sur quelques principes

Objectif : introduire un modèle d'analyse du temps théorique d'exécution des algorithmes

Ce modèle doit :

- s'affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)
- être utilisable pour évaluer au mieux les temps d'exécution des programmes
  (programme = algorithme écrit dans un langage de programmation)
- envisager tous les cas de figures possibles pour les données en entrée
- considérer l'accroissement de la taille de la donnée en entrée

Un modèle d’analyse fondé sur quelques principes

Objectif : introduire un modèle d’analyse du temps théorique d’exécution des algorithmes

Ce modèle doit :

- s’affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)
- être utilisable pour évaluer au mieux les temps d’exécution des programmes
  (programme = algorithme écrit dans un langage de programmation)
- envisager tous les cas de figures possibles pour les données en entrée
- considérer l’accroissement de la taille de la donnée en entrée

Ce modèle d’analyse est fondé sur quelques principes simples (la suite)

Principe 1 : Actions élémentaires en temps constant

Principe 1 : Actions élémentaires exécutables en temps constant

On identifie 4 types d'actions élémentaires :

- **Affectations :** par exemple **y = x;** dont le coût d'exécution est une constante $k_a \in \mathbb{R}^{*+}$ avec
  - $k_a \neq 0$: le temps peut être tres petit mais il ne peut être nul
  - $k_a > 0$: le temps ne peut être négatif

Principe 1 : Actions élémentaires en temps constant

Principe 1 : Actions élémentaires exécutables en temps constant

On identifie 4 types d'actions élémentaires :

- **Affectations :** par exemple **y = x;** dont le coût d'exécution est une constante $k_a \in \mathbb{R}^{*+}$ avec
    - $k_a \neq 0$: le temps peut être tres petit mais il ne peut être nul
    - $k_a > 0$: le temps ne peut être négatif
- **Opérations arithmétiques ou logiques :** par exemple l'addition dans **x = a+n;** de coût $k_o \in \mathbb{R}^{*+}$

Principe 1 : Actions élémentaires en temps constant

Principe 1 : Actions élémentaires exécutables en temps constant

On identifie 4 types d'actions élémentaires :

- **Affectations :** par exemple **y = x;** dont le coût d'exécution est une constante $k_a \in \mathbb{R}^{*+}$ avec
  - $k_a \neq 0$: le temps peut être tres petit mais il ne peut être nul
  - $k_a > 0$: le temps ne peut être négatif
- **Opérations arithmétiques ou logiques :** par exemple l'addition dans **x = a+n;** de coût $k_o \in \mathbb{R}^{*+}$
- **Tests :** par exemple **n** $<$ **2** de coût $k_t \in \mathbb{R}^{*+}$

Principe 1 : Actions élémentaires en temps constant

Principe 1 : Actions élémentaires exécutables en temps constant

On identifie 4 types d'actions élémentaires :

- **Affectations :** par exemple **y = x;** dont le coût d'exécution est une constante $k_a \in \mathbb{R}^{*+}$ avec
  - $k_a \neq 0$: le temps peut être tres petit mais il ne peut être nul
  - $k_a > 0$: le temps ne peut être négatif
- **Opérations arithmétiques ou logiques :** par exemple l'addition dans **x = a+n;** de coût $k_o \in \mathbb{R}^{*+}$
- **Tests :** par exemple **n** $<$ **2** de coût $k_t \in \mathbb{R}^{*+}$
- **Branchements :** dans le cadre d'une instruction conditionnelle ou d'une boucle, il est opéré un branchement vers l'alternative d'un **if** par exemple ou vers une sortie de boucle ; son coût est $k_b \in \mathbb{R}^{*+}$

On dit parfois *opérations fondamentales* à la place d'actions élémentaires

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1}:

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1}:

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1}:

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)
- 1 opération arithmétique : coût k_o

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1}:

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit $\mathbf{x = 2015;}$ est exécutée, soit $\mathbf{y = x;}$)
- 1 opération arithmétique : coût k_o
- 2 tests : coût $2k_t$
  (celui de $\mathbf{n} < \mathbf{2}$ et celui de la comparaison à 0 ou 1 de la condition du **if**)

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1}:

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)
- 1 opération arithmétique : coût k_o
- 2 tests : coût $2k_t$
  (celui de **n** $<$ **2** et celui de la comparaison à 0 ou 1 de la condition du **if**)
- 1 branchement : coût k_b
  (1 branchement après **n** $<$ **2** ou après **x = 2015;**)

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1}:

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)
- 1 opération arithmétique : coût k_o
- 2 tests : coût $2k_t$
  (celui de **n** $<$ **2** et celui de la comparaison à 0 ou 1 de la condition du **if**)
- 1 branchement : coût k_b
  (1 branchement après **n** $<$ **2** ou après **x = 2015;**)

Complexité de A_1 **:** $T_{A_1} = 5k_a + k_o + 2k_t + k_b$ soit un temps constant
(la somme de 9 constantes est une constante)

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés (car liés à des tests)
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

Attention : on verra plus loin que le modèle peut poser souci...

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés (car liés à des tests)
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

  Attention : on verra plus loin que le modèle peut poser souci...
- **L'analyse est totalement indépendante des machines utilisables**
  ramener le temps d'exécution d'une action élémentaire à une constante $k \in \mathbb{R}^{*+}$ permet de s'affranchir des questions de matériel

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés (car liés à des tests)
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

  Attention : on verra plus loin que le modèle peut poser souci...
- **L'analyse est totalement indépendante des machines utilisables**
  ramener le temps d'exécution d'une action élémentaire à une constante $k \in \mathbb{R}^{*+}$ permet de s'affranchir des questions de matériel
- k **sera précisé en pratique** selon que l'on utilise un "vieux" PC ou le supercalculateur Fugaku qui tourne à 442 péta FLOPS...

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés (car liés à des tests)
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

  Attention : on verra plus loin que le modèle peut poser souci...
- **L'analyse est totalement indépendante des machines utilisables** ramener le temps d'exécution d'une action élémentaire à une constante $k \in \mathbb{R}^{*+}$ permet de s'affranchir des questions de matériel
- k **sera précisé en pratique** selon que l'on utilise un "vieux" PC ou le supercalculateur Fugaku qui tourne à 442 péta FLOPS...
- **Et l'algorithme est écrit en C car 76 fois moins énergivore que Python** ... merci pour la planète

Principe 2 : Analyse en fonction de la donnée en entrée.

L'analyse s'opère en fonction de la taille de la donnée en entrée

- **Évaluation du temps d'exécution :** en fonction de la taille de la donnée à traiter

  (Naturel : a priori, le temps de calcul augmente avec une donnée plus grande...)

Principe 2 : Analyse en fonction de la donnée en entrée.

L'analyse s'opère en fonction de la taille de la donnée en entrée

- **Évaluation du temps d'exécution :** en fonction de la taille de la donnée à traiter
  (Naturel : a priori, le temps de calcul augmente avec une donnée plus grande...)
- **Taille de la donnée en entrée ?**
  - test de primalité ($n \in \mathbb{N}$ premier ?) : taille du codage de l'entier
  - algorithme de tri : taille du tableau à trier
  - algorithme de graphe : selon la représentation listes ou matrices

Principe 2 : Analyse en fonction de la donnée en entrée.

L'analyse s'opère en fonction de la taille de la donnée en entrée

- **Évaluation du temps d'exécution :** en fonction de la taille de la donnée à traiter

  (Naturel : a priori, le temps de calcul augmente avec une donnée plus grande...)

- **Taille de la donnée en entrée ?**
  - test de primalité ($n \in \mathbb{N}$ premier ?) : taille du codage de l'entier
  - algorithme de tri : taille du tableau à trier
  - algorithme de graphe : selon la représentation listes ou matrices

- **Incidence formelle**
  - complexité d'un algorithme A : une fonction $T_A : \mathbb{N} \rightarrow \mathbb{R}^+$
  - avec une donnée de taille n, par exemple $T_A(n) = 7k.n^2 + 3kn + 24k$.

Principe 3 : Comportement asymptotique

L'analyse prend en compte le comportement asymptotique du temps

- **Un ordinateur est fait pour traiter des grands jeux de données**

Principe 3 : Comportement asymptotique

L'analyse prend en compte le comportement asymptotique du temps

- **Un ordinateur est fait pour traiter des grands jeux de données**
- **Comment évolue le temps quand la taille de la donnée s'accroît**

  Exploitation d'un programme sur de plus grands jeux de données que ceux utilisés pour sa mise au point

  (Rappel : le tri à bulles est 100 fois plus lent sur un tableau 10 fois plus grand...)

Principe 3 : Comportement asymptotique

L'analyse prend en compte le comportement asymptotique du temps

- **Un ordinateur est fait pour traiter des grands jeux de données**
- **Comment évolue le temps quand la taille de la donnée s'accroît**

  Exploitation d'un programme sur de plus grands jeux de données que ceux utilisés pour sa mise au point

  (Rappel : le tri à bulles est 100 fois plus lent sur un tableau 10 fois plus grand...)
- **Comportement asymptotique du temps**

  Étude de la fonction $T_A : \mathbb{N} \to \mathbb{R}^+$ quand n tend vers l'infini

  (même si aucun jeu de données ne sera de taille infinie en pratique... *a priori*)

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

En sortie **present** (variable de nature logique, vrai / faux) est affectée

- à vrai soit 1 si l'entier **x** est mémorisé dans le tableau **t** ;
- sinon à faux soit 0.

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$ (*on simplifie encore le modèle...*)
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$ (*on simplifie encore le modèle...*)
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$ (*on simplifie encore le modèle...*)
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$
- affectation de la variable **present** à 1 : peut coûter de 0 à $k.n$
  selon que **x** n'apparaît pas dans le tableau **t** ou figure dans chaque case

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test $\mathbf{i} < \mathbf{n}$: coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$
- affectation de la variable **present** à 1 : peut coûter de 0 à $k.n$
  selon que **x** n'apparaît pas dans le tableau **t** ou figure dans chaque case

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$
- affectation de la variable **present** à 1 : peut coûter de 0 à $k.n$
  selon que **x** n'apparaît pas dans le tableau **t** ou figure dans chaque case

Complexité de A_2 :

$$2k + 2k.n + (k.n + k) + k.n = 3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$$

soit

$$3k + 4k.n \leq T_{A_2(n)} \leq 3k + 5k.n$$

(encadrement du fait de l'incertitude sur la présence de la valeur **x** dans **t**)

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2}(n)$ est **linéaire**

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2}(n)$ est **linéaire**
- $4k$ ou $5k$ sont juste des **constantes multiplicatives**
  car elles n'influent pas sur la forme de l'accroissement du temps
  (on dit parfois **constantes cachées** si on ne les évoque pas)

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2}(n)$ est **linéaire**
- $4k$ ou $5k$ sont juste des **constantes multiplicatives** car elles n'influent pas sur la forme de l'accroissement du temps (on dit parfois **constantes cachées** si on ne les évoque pas)
- Comportement linéaire de la fonction $T_{A_2}(n)$:
    - la complexité est en $\mathrm{O}(n)$ pour parler de **majoration** du temps de calcul ce sera noté $T_{A_2(n)} \in \mathrm{O}(n)$ (prononcer "*en grand o de n*")
    - la complexité est en $\Theta(n)$ pour parler **d'ordre exact** du temps de calcul ce sera noté $T_{A_2(n)} \in \Theta(n)$ (prononcer "*en thêta de n*")

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2}(n)$ est **linéaire**
- $4k$ ou $5k$ sont juste des **constantes multiplicatives** car elles n'influent pas sur la forme de l'accroissement du temps (on dit parfois **constantes cachées** si on ne les évoque pas)
- Comportement linéaire de la fonction $T_{A_2}(n)$:
  - la complexité est en $O(n)$ pour parler de **majoration** du temps de calcul ce sera noté $T_{A_2(n)} \in O(n)$ (prononcer "*en grand o de n*")
  - la complexité est en $\Theta(n)$ pour parler **d'ordre exact** du temps de calcul ce sera noté $T_{A_2(n)} \in \Theta(n)$ (prononcer "*en thêta de n*")

Mais il y a 2 soucis :

- Encadrement du temps ⇒ incertitude, imprécision
- Et qui écrirait un tel algorithme ?

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas : on considère un jeu de données qui maximise le temps de calcul de l'algorithme

- permet d'obtenir des garanties car ce temps ne sera jamais dépassé
- permet en général d'améliorer l'approche en évitant les pires cas

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas : on considère un jeu de données qui maximise le temps de calcul de l'algorithme

- permet d'obtenir des garanties car ce temps ne sera jamais dépassé
- permet en général d'améliorer l'approche en évitant les pires cas

Retour sur l'algorithme A_2 :

- on avait trouvé : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$
- on lève l'incertitude : $T_{A_2}(n) = 3k + 5k.n$

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas : on considère un jeu de données qui maximise le temps de calcul de l'algorithme

- permet d'obtenir des garanties car ce temps ne sera jamais dépassé
- permet en général d'améliorer l'approche en évitant les pires cas

Retour sur l'algorithme A_2 :

- on avait trouvé : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$
- on lève l'incertitude : $T_{A_2}(n) = 3k + 5k.n$

Améliorer A_2 en évitant les pire cas ?

Idée : arrêter l'exécution dès que **x** est trouvé dans **t** !

On regarde cela, mais avant quelques remarques

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
  - mise en évidence d'un modèle probabiliste pour les jeux de données → pas toujours simple, et parfois impossible (cf. n'a aucun sens)
  - analyse souvent très complexe
  - pas pertinent dans cette UE...

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
    - mise en évidence d'un modèle probabiliste pour les jeux de données → pas toujours simple, et parfois impossible (cf. n'a aucun sens)
    - analyse souvent très complexe
    - pas pertinent dans cette UE...

- analyse **lisse** d'algorithme (*smoothed analysis*)
    - pour éviter certains soucis de l'analyse en moyenne
    - pas pertinent dans cette UE...

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
    - mise en évidence d'un modèle probabiliste pour les jeux de données
      → pas toujours simple, et parfois impossible (cf. n'a aucun sens)
    - analyse souvent très complexe
    - pas pertinent dans cette UE...
- analyse **lisse** d'algorithme (*smoothed analysis*)
    - pour éviter certains soucis de l'analyse en moyenne
    - pas pertinent dans cette UE...
- analyse de la complexié dans le **meilleur des cas**
    - pour les seuls optimiste ? Non, c'est parfois utile pour faciliter l'analyse
    - pas pertinent dans cette UE...

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
    - mise en évidence d'un modèle probabiliste pour les jeux de données → pas toujours simple, et parfois impossible (cf. n'a aucun sens)
    - analyse souvent très complexe
    - pas pertinent dans cette UE...
- analyse **lisse** d'algorithme (*smoothed analysis*)
    - pour éviter certains soucis de l'analyse en moyenne
    - pas pertinent dans cette UE...
- analyse de la complexié dans le **meilleur des cas**
    - pour les seuls optimiste ? Non, c'est parfois utile pour faciliter l'analyse
    - pas pertinent dans cette UE...

Dans cette UE : focalisation sur l'analyse dans le pire des cas

Principe 4 : Pire des cas (dans cette UE)

Exemple : A_3 résout (plus efficacement ?) le même problème que A_2

```
present = 0; i = 0;
while( !present && (i < n) )
      if ( t[i] == x ) present = 1; else i = i+1;
```

Principe 4 : Pire des cas (dans cette UE)

Exemple : A_3 résout (plus efficacement ?) le même problème que A_2

```
present = 0; i = 0;
while( !present && (i < n) )
     if ( t[i] == x ) present = 1; else i = i+1;
```

Analyse : conduit à trouver $T_{A_3}(n) = 6k + 7k.n$

- identification du pire des cas : donnée maximisant le temps de calcul
    - meilleur des cas : $\mathbf{x = t[0]}$ car arrêt immédiat ⇒ coûte $10k$
    - pire des cas : **x** n'est pas dans **t** ⇒ tout le tableau est parcouru

Principe 4 : Pire des cas (dans cette UE)

Exemple : A_3 résout (plus efficacement ?) le même problème que A_2

```
present = 0; i = 0;
while( !present && (i < n) )
      if ( t[i] == x ) present = 1; else i = i+1;
```

Analyse : conduit à trouver $T_{A_3}(n) = 6k + 7k.n$

- identification du pire des cas : donnée maximisant le temps de calcul
  - meilleur des cas : **x = t[0]** car arrêt immédiat ⇒ coûte $10k$
  - pire des cas : **x** n'est pas dans **t** ⇒ tout le tableau est parcouru
- analyse dans le pire des cas (avec n passages dans la boucle)
  - n tests d'entée positifs et 1 test négatif (chacun coûte $4k$) (négation **!** + test **!present** + test **i<n** + conjonction **&&**)
  - n tests d'égalité **t[i] == x** négatifs (chacun coûte k)
  - n incrémentations de **i** (chacune coûte $2k$)

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

  $$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

  et c'est même pire !!!

  (même si "en général" A_3 sera plus efficace que A_2)

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

  $$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

  et c'est même pire !!!

  (même si "en général" A_3 sera plus efficace que A_2)
- analyse dans le pire des cas → offre une garantie :
  on ne fera jamais pire !

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

$$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

  et c'est même pire !!!

  (même si "en général" A_3 sera plus efficace que A_2)

- analyse dans le pire des cas → offre une garantie :
  on ne fera jamais pire !
- cette analyse permet de lever les doutes précédents

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

  $$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

  et c'est même pire !!!

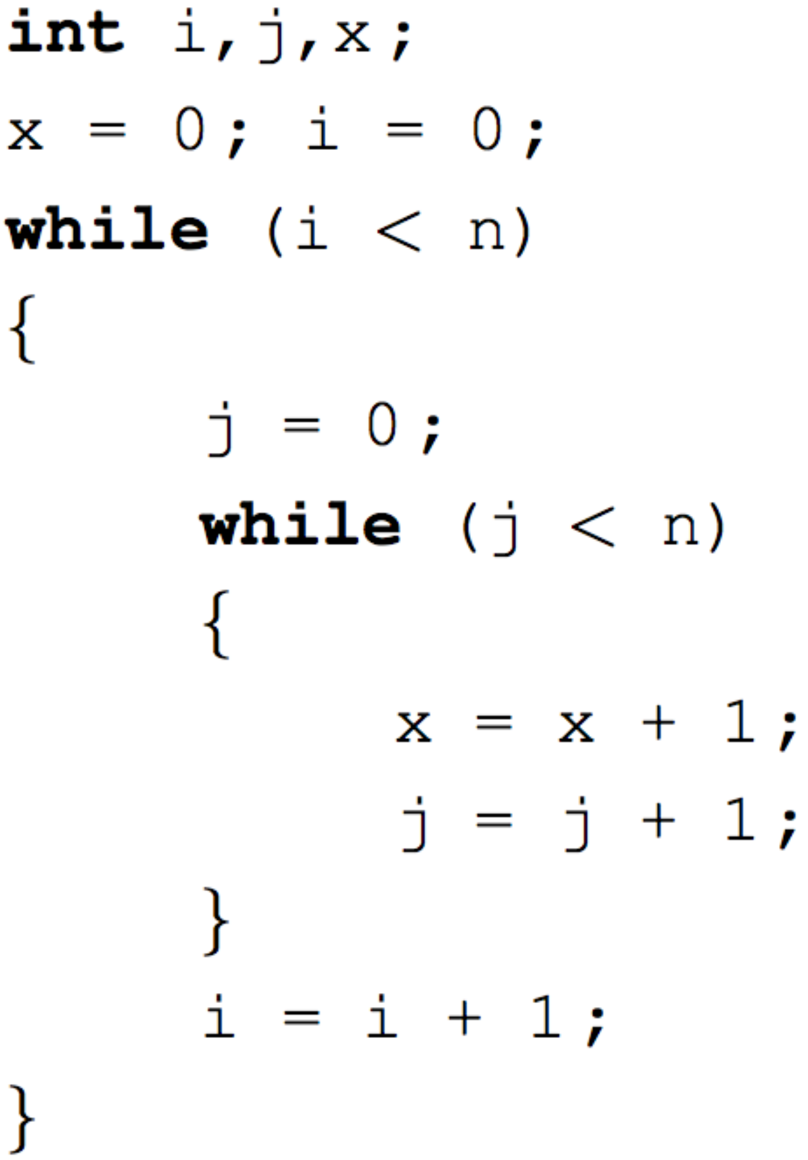
  (même si "en général" A_3 sera plus efficace que A_2)
- analyse dans le pire des cas → offre une garantie :
  on ne fera jamais pire !
- cette analyse permet de lever les doutes précédents

Dans cette UE, on ne s'intéressera qu'au pire des cas car :

- pour des problème difficiles on peut avoir des cas triviaux
  (si l'aiguille cherchée dans la botte de foin brille à l'entrée du tas...)
- un problème sera jugé difficile même s'il recèle "peu" de cas difficile

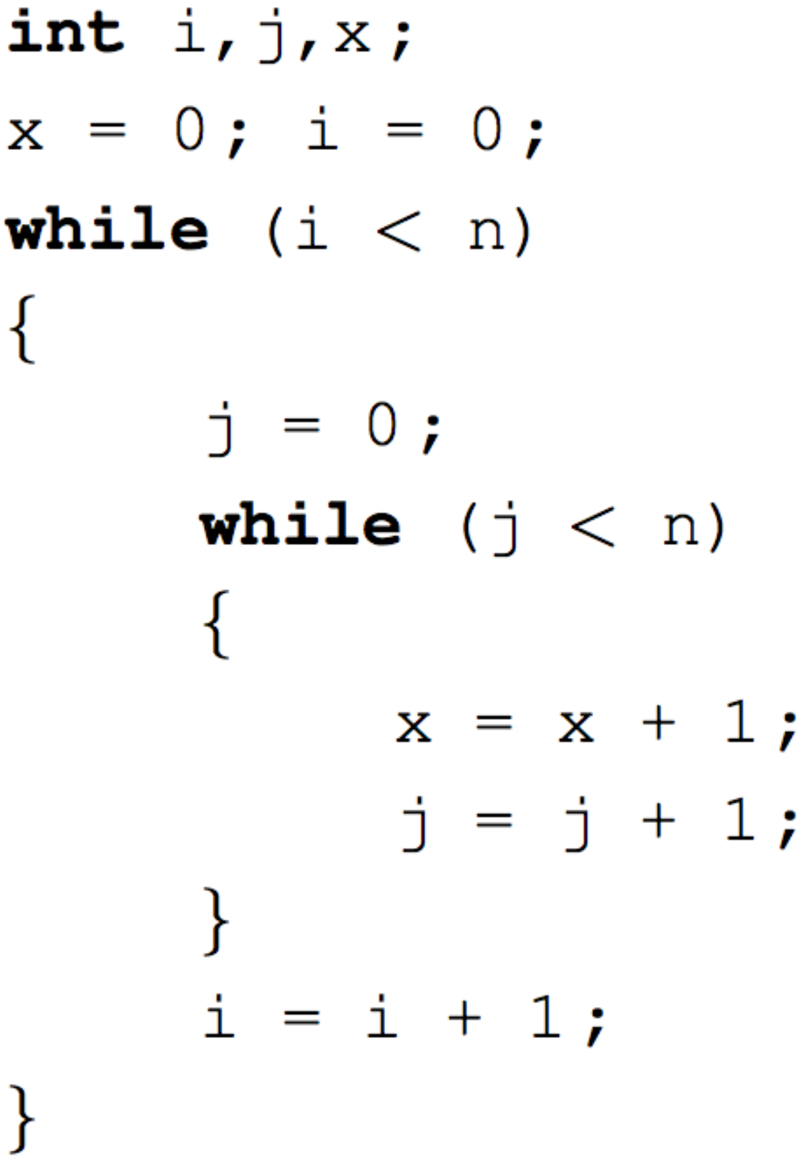
Plan

Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < n)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Algorithme A_4 : avec des boucles imbriquées



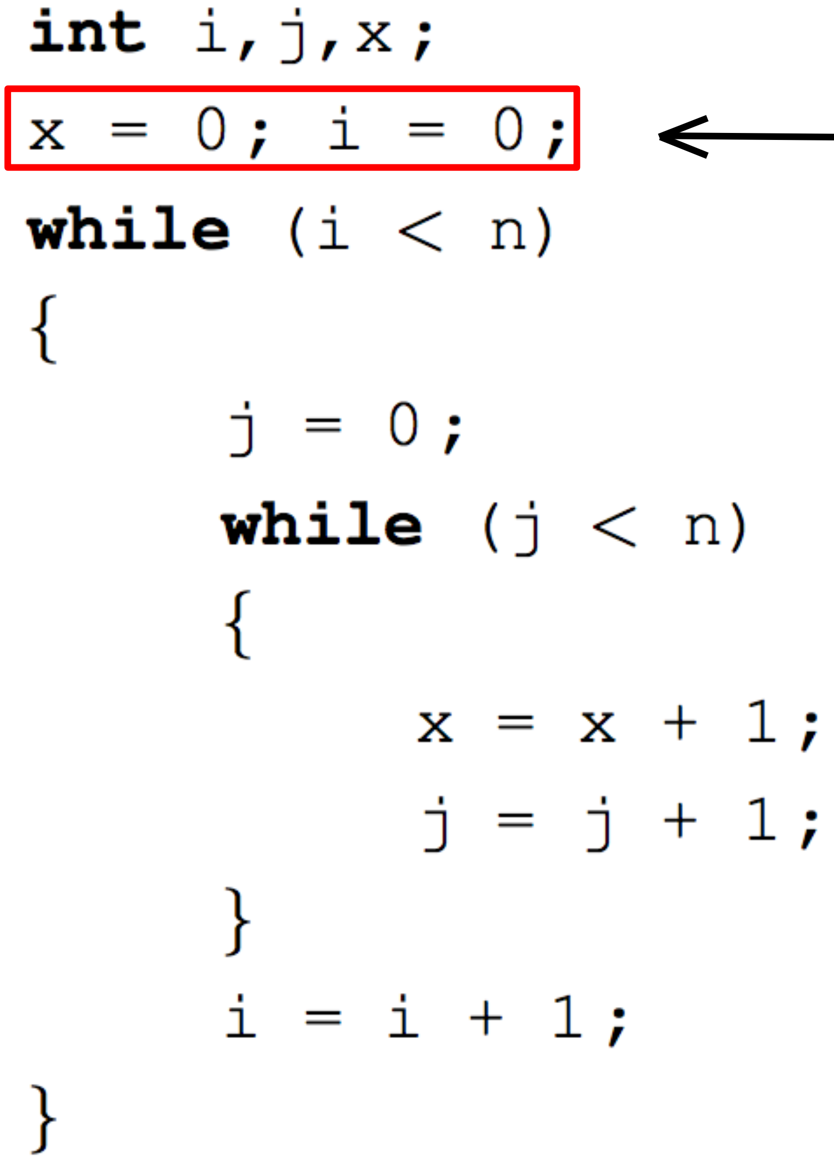
```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < n)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Pire de cas ?

En fait, il n'y a ici qu'un seul cas !

On dit alors parfois : « dans tous les cas »

Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < n)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

coûte $2k$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
     j = 0;
     while (j < n)
     {
          x = x + 1;
          j = j + 1;
     }
     i = i + 1;
}
```

`x = 0; i = 0;` ← coûte $2k$

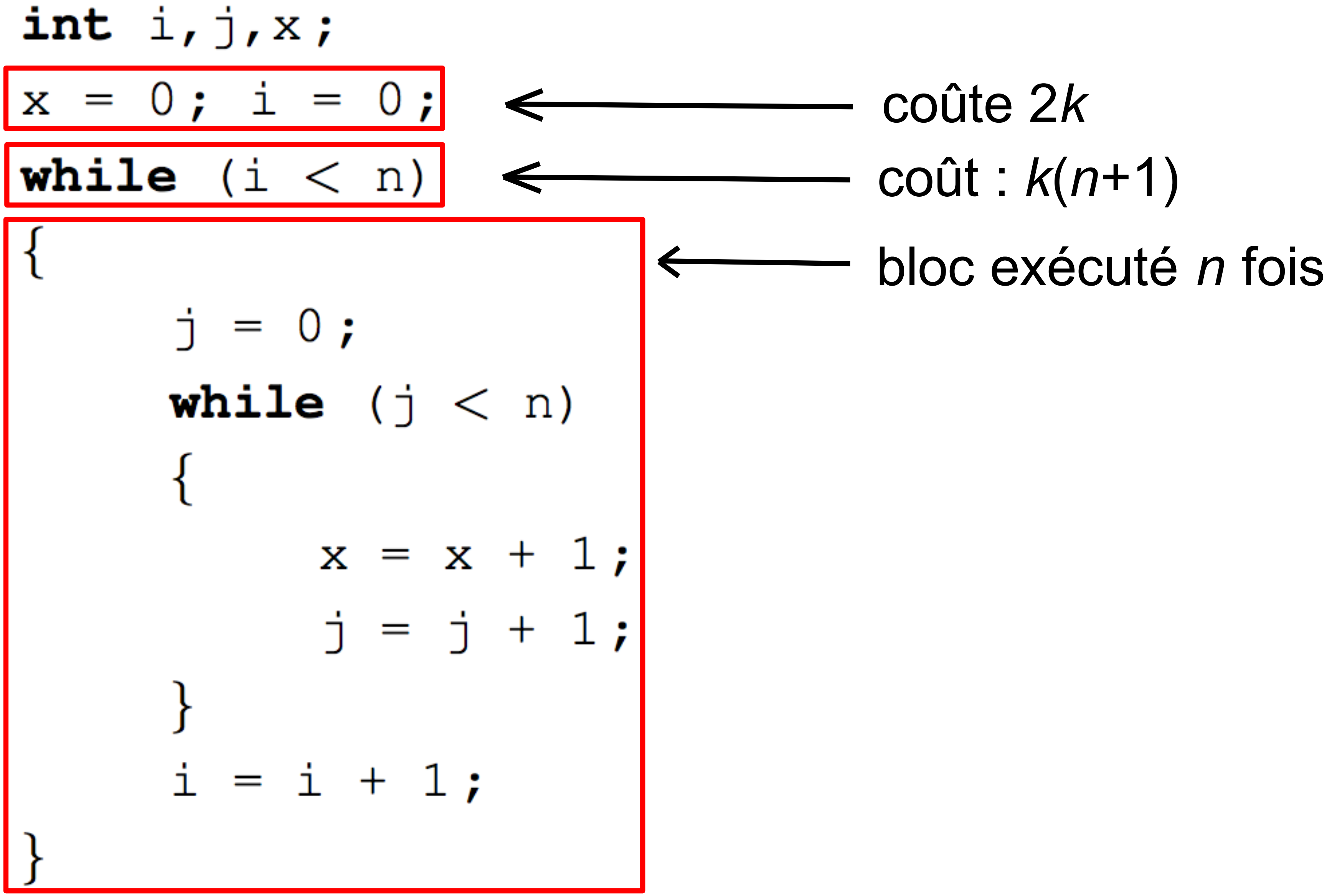
`while (i < n)` ← coût : $k(n+1)$

(contrôle de la boucle)

n tests positifs : i=0,... i=n-1

1 test négatif : i=n

Algorithme A_4 : avec des boucles imbriquées



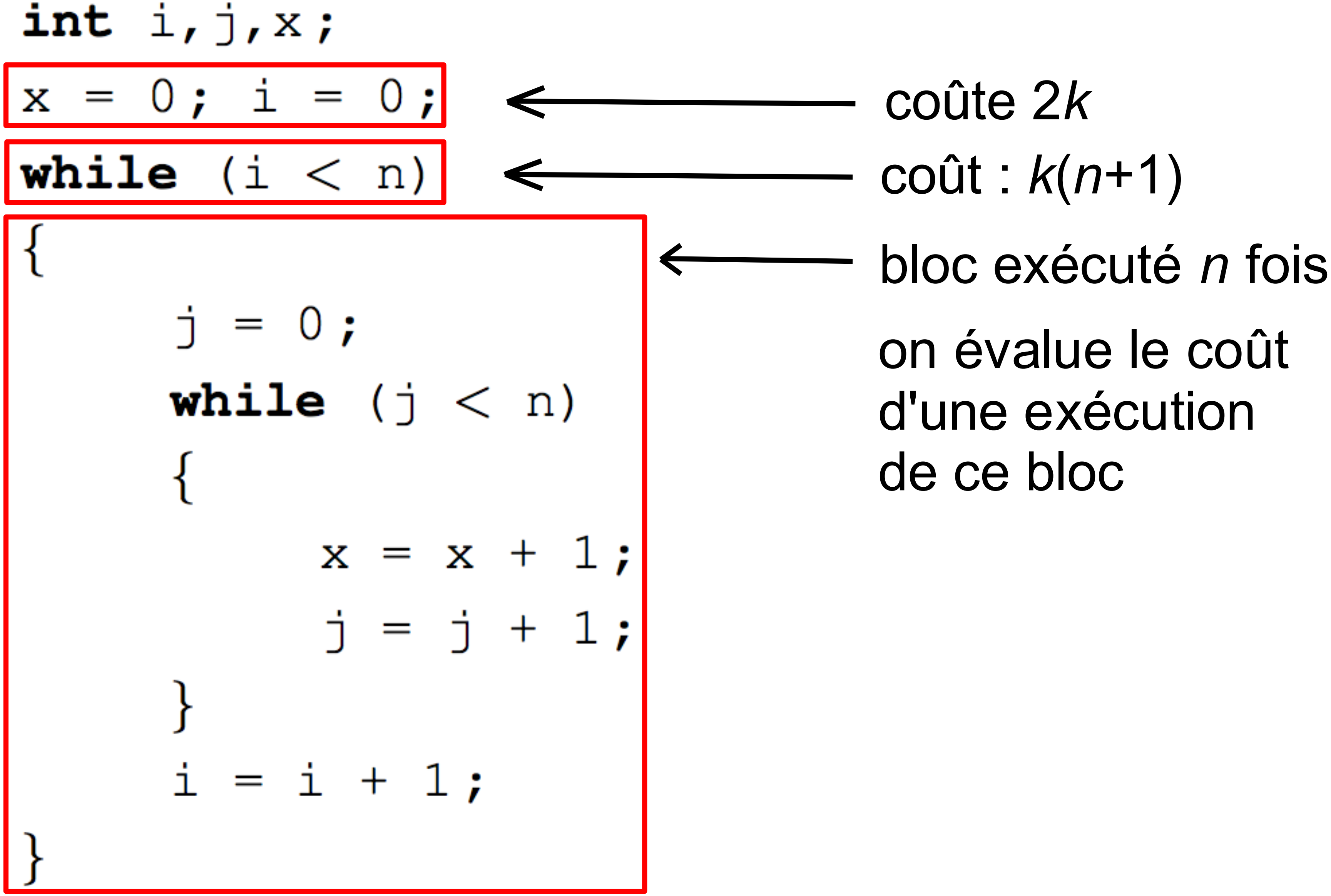
```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

coûte $2k$

coût : $k(n+1)$

bloc exécuté n fois

Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < n)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

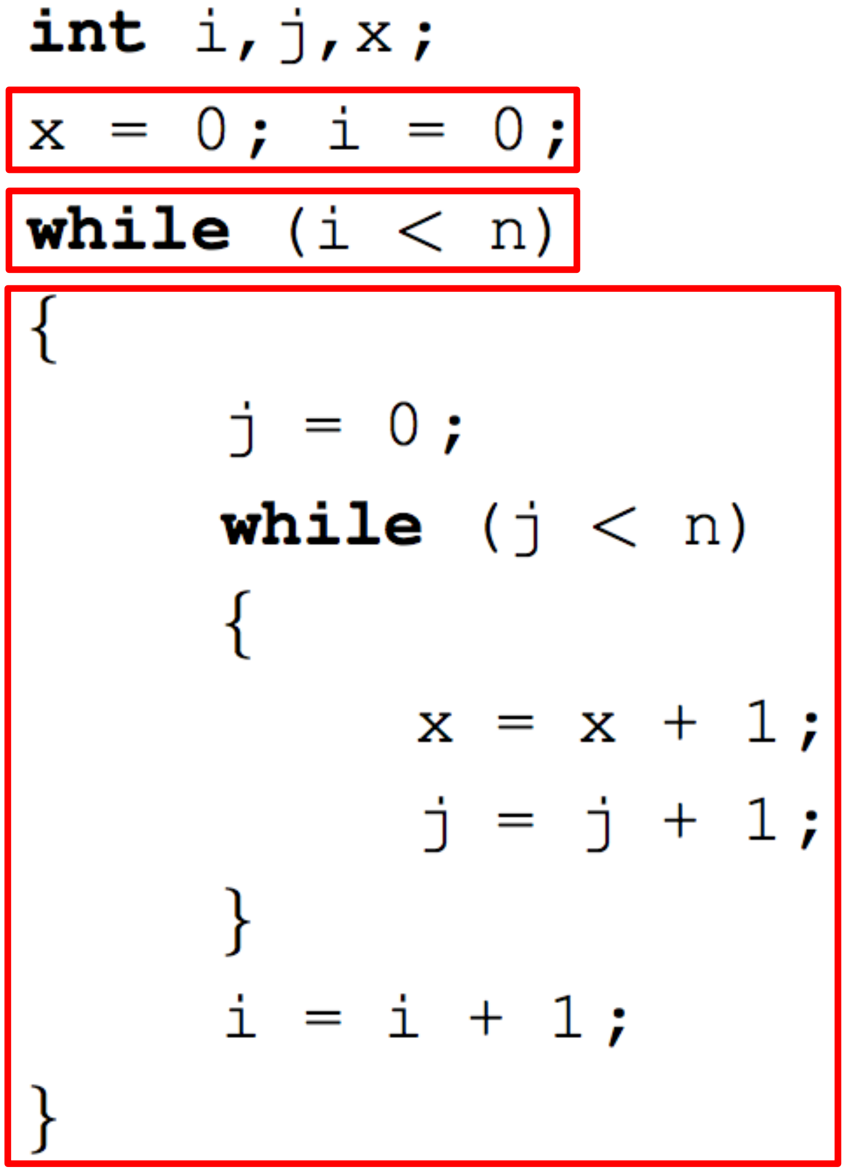
coûte $2k$

coût : $k(n+1)$

bloc exécuté n fois

on évalue le coût
d'une exécution
de ce bloc

Algorithme A_4 : avec des boucles imbriquées

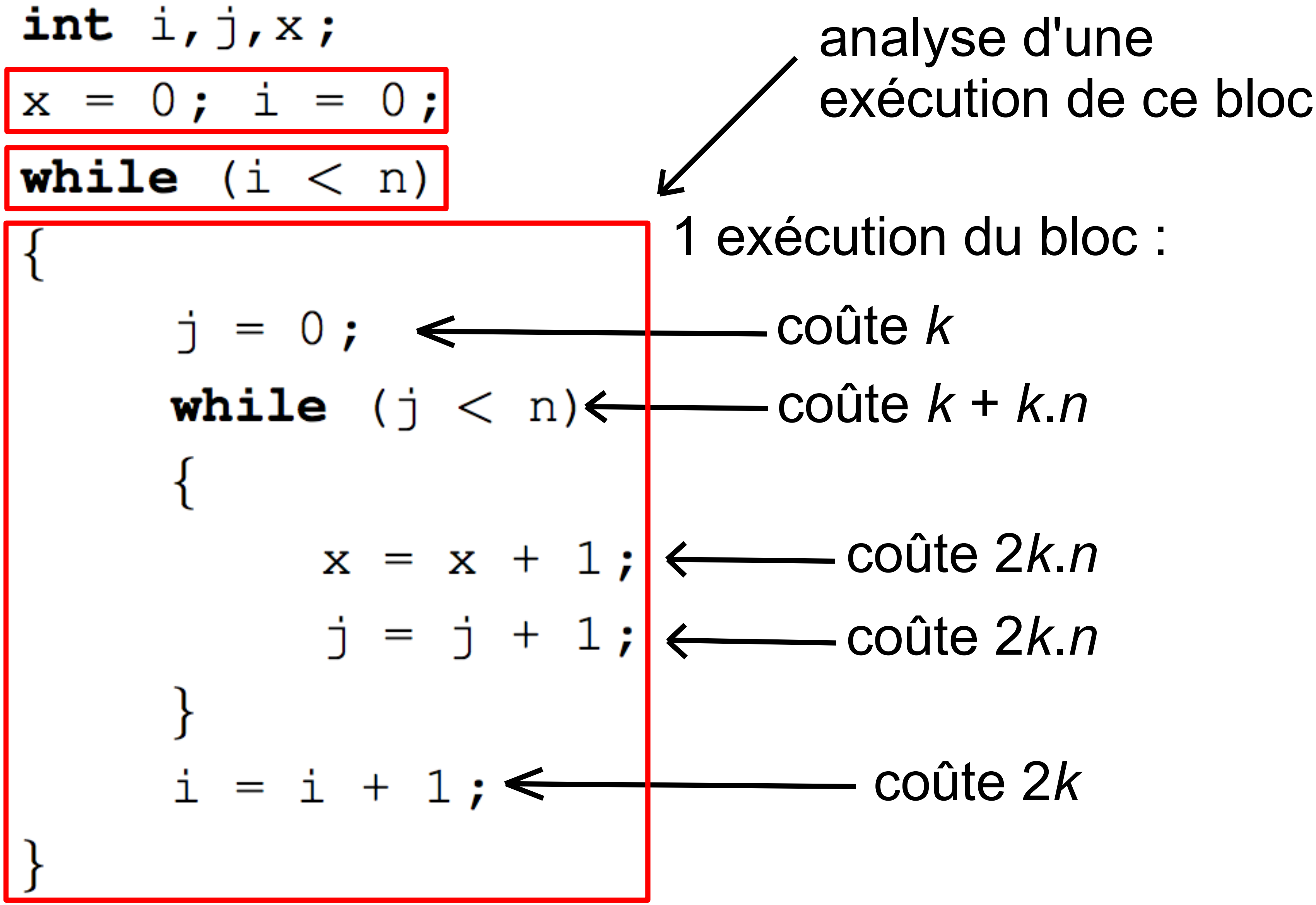


```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
     j = 0;
     while (j < n)
     {
          x = x + 1;
          j = j + 1;
     }
     i = i + 1;
}
```

analyse d'une exécution de ce bloc

1 exécution du bloc ?

Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

analyse d'une exécution de ce bloc

1 exécution du bloc :

coûte k

coûte $k + k.n$

coûte $2k.n$

coûte $2k.n$

coûte $2k$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

analyse d'une exécution de ce bloc

1 exécution du bloc : $4k + 5k.n$

`j = 0;` coûte k

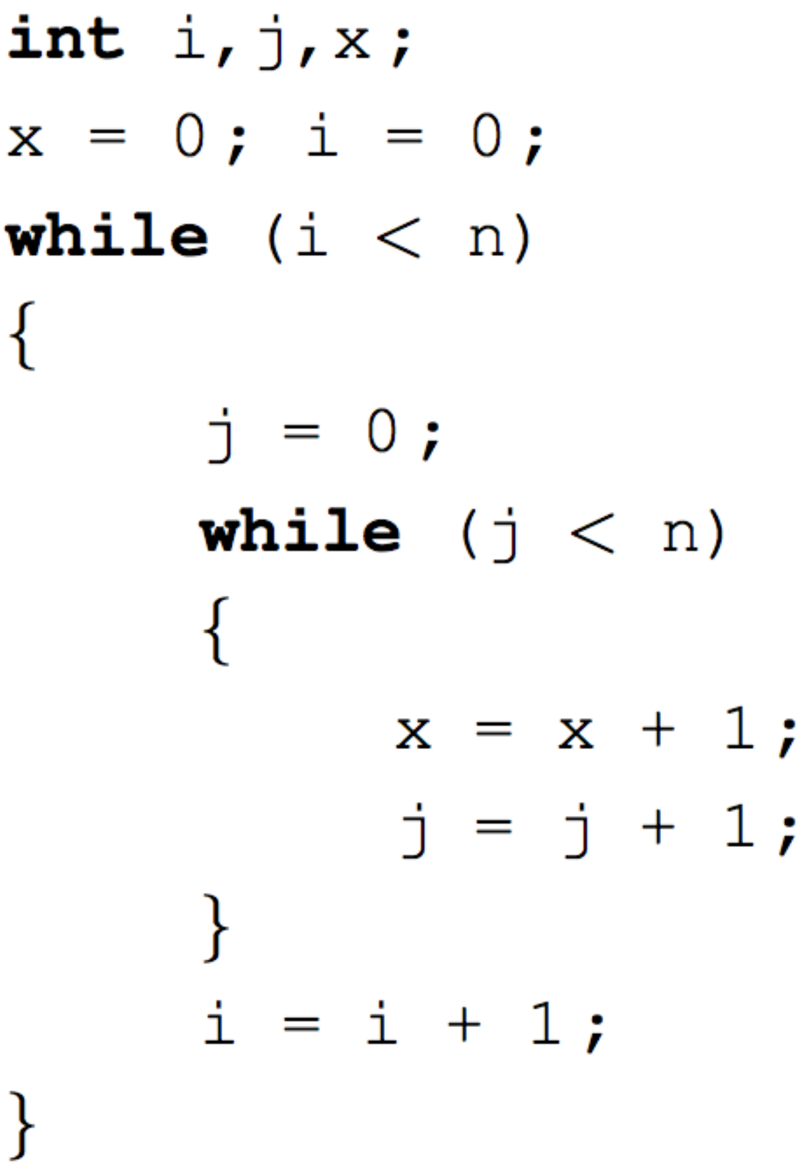
`while (j < n)` coûte $k + k.n$

`x = x + 1;` coûte $2k.n$

`j = j + 1;` coûte $2k.n$

`i = i + 1;` coûte $2k$

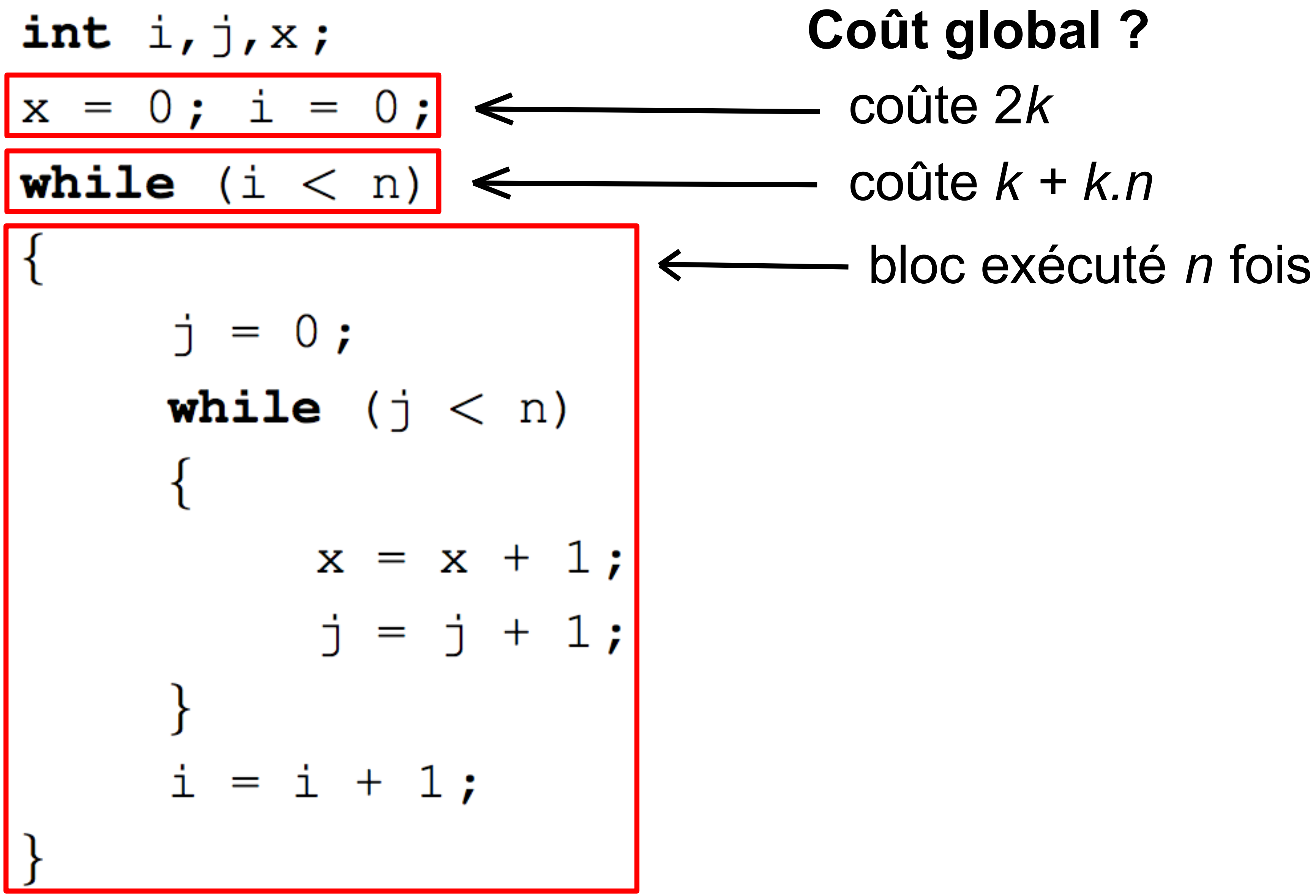
Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < n)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Coût global ?

Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

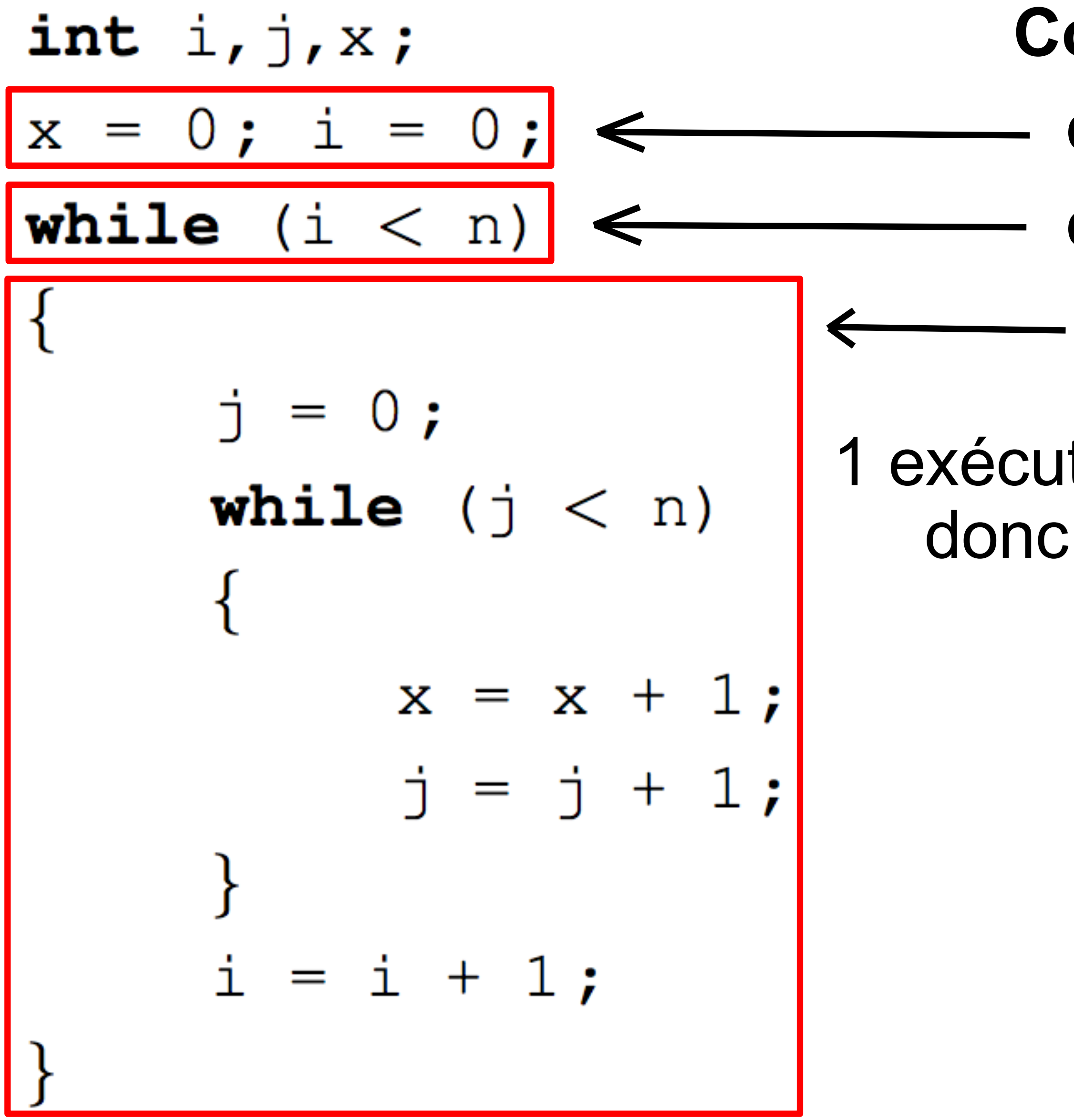
Coût global ?

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Coût global ?

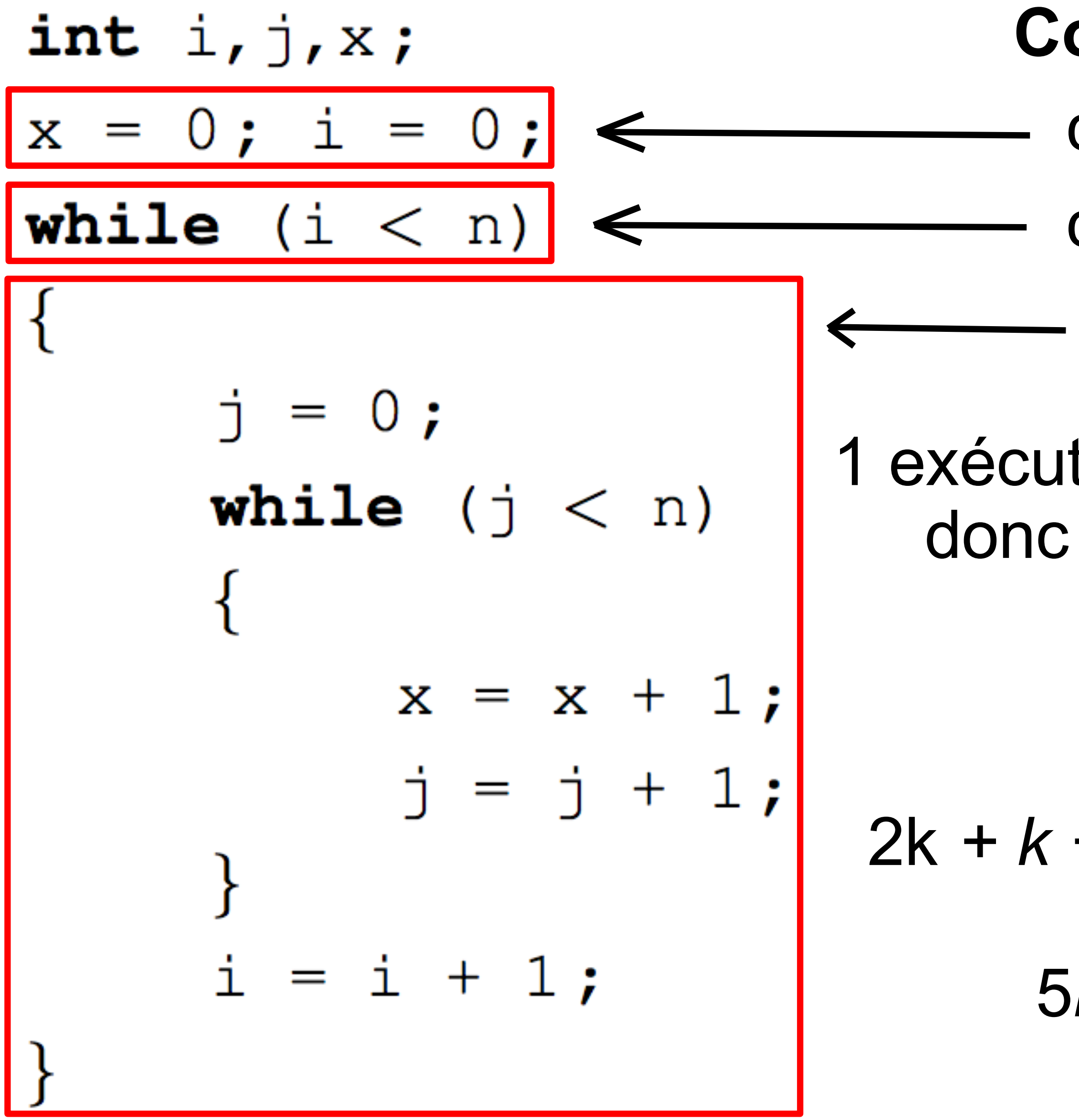
coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

1 exécution du bloc : $4k + 5k.n$
donc pour n exécutions :
n x $(4k + 5k.n)$

Algorithme A_4 : avec des boucles imbriquées



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < n)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Coût global ?

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

1 exécution du bloc : $4k + 5k.n$
donc pour n exécutions :
n x $(4k + 5k.n)$

Coût global :

$$2k + k + k.n + n \times (4k + 5k.n)$$
$$=$$
$$5k.n^2 + 5k.n + 3k$$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < n)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Coût global :

$T_{A_4}(n) = k(5.n^2 + 5n + 3)$

Complexité :

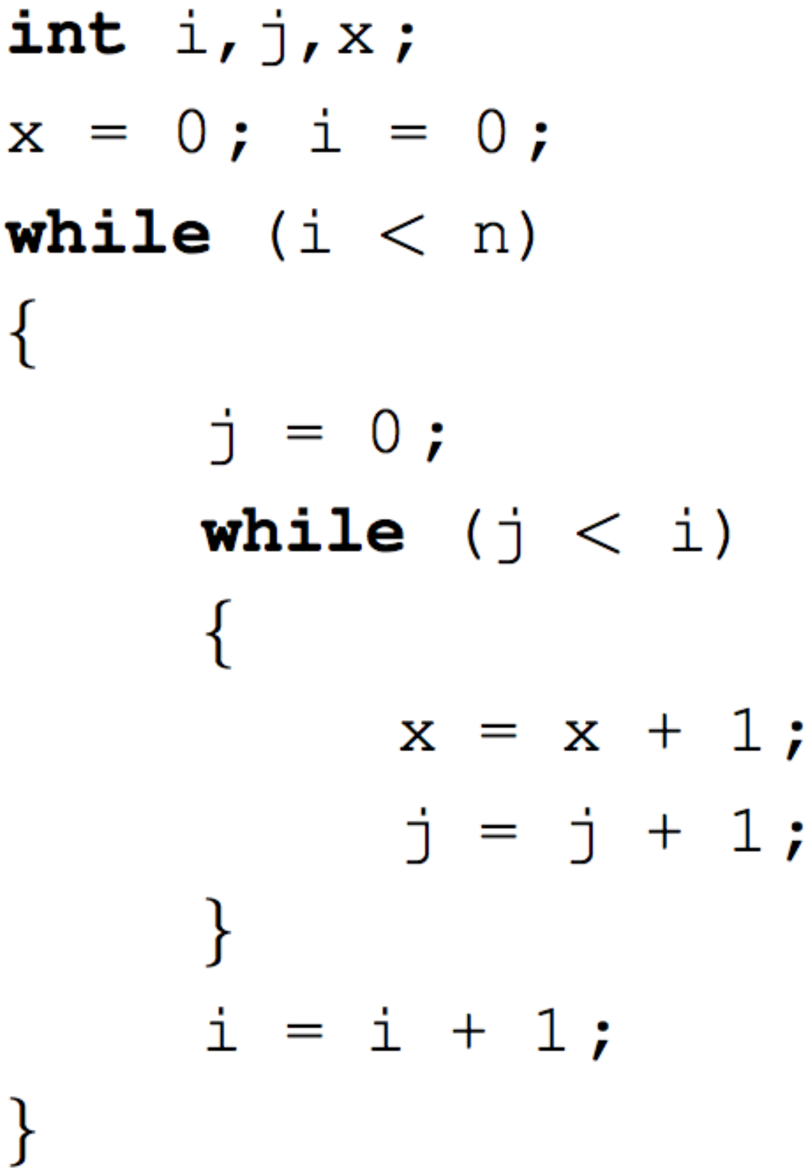
A_4 est donc en $O(n^2)$
car $5k.n$ et $3k$ sont négligeables
par rapport à $5k.n^2$
(O : majoration)

et pour l'ordre exact

A_4 est donc en $\Theta(n^2)$

Algorithme A_5 : une variation sur A_4

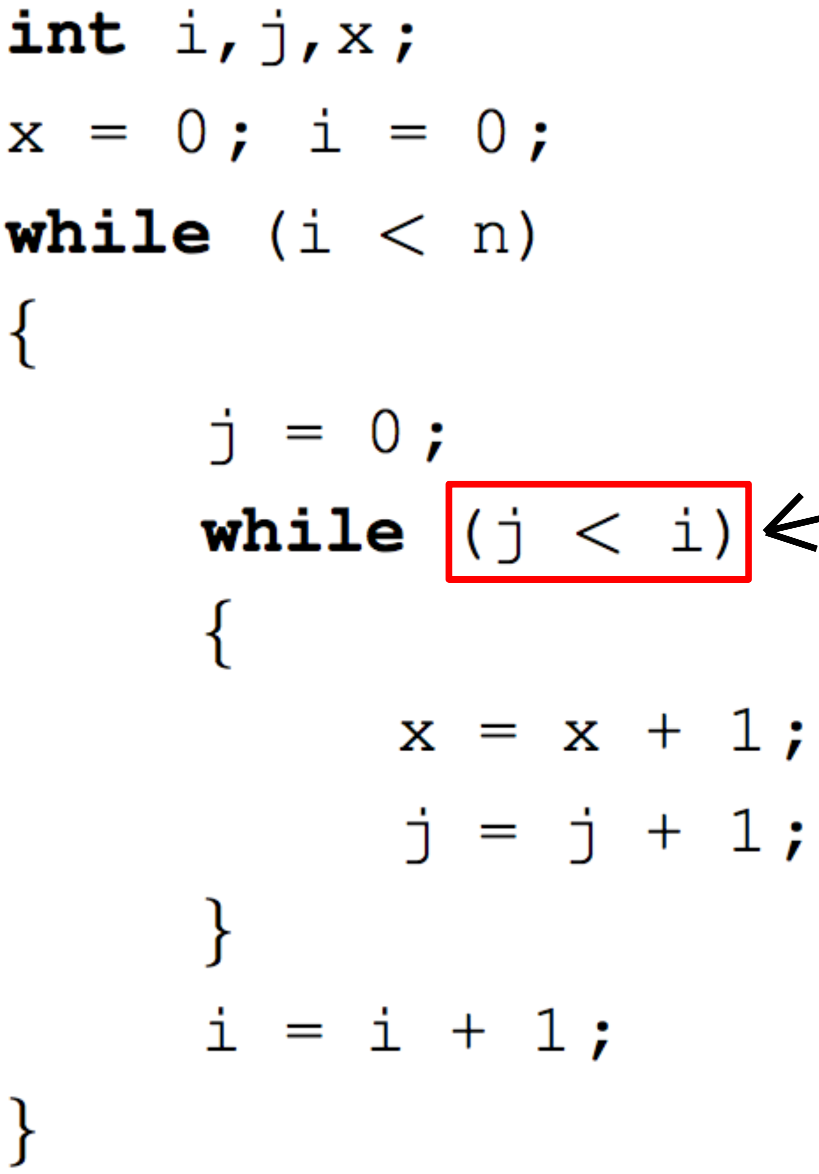
Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Juste une petite modification :

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
     j = 0;
     while (j < i)
     {
          x = x + 1;
          j = j + 1;
     }
     i = i + 1;
}
```

Juste une petite modification :
le test
(j < n)
est remplacé par
(j < i)

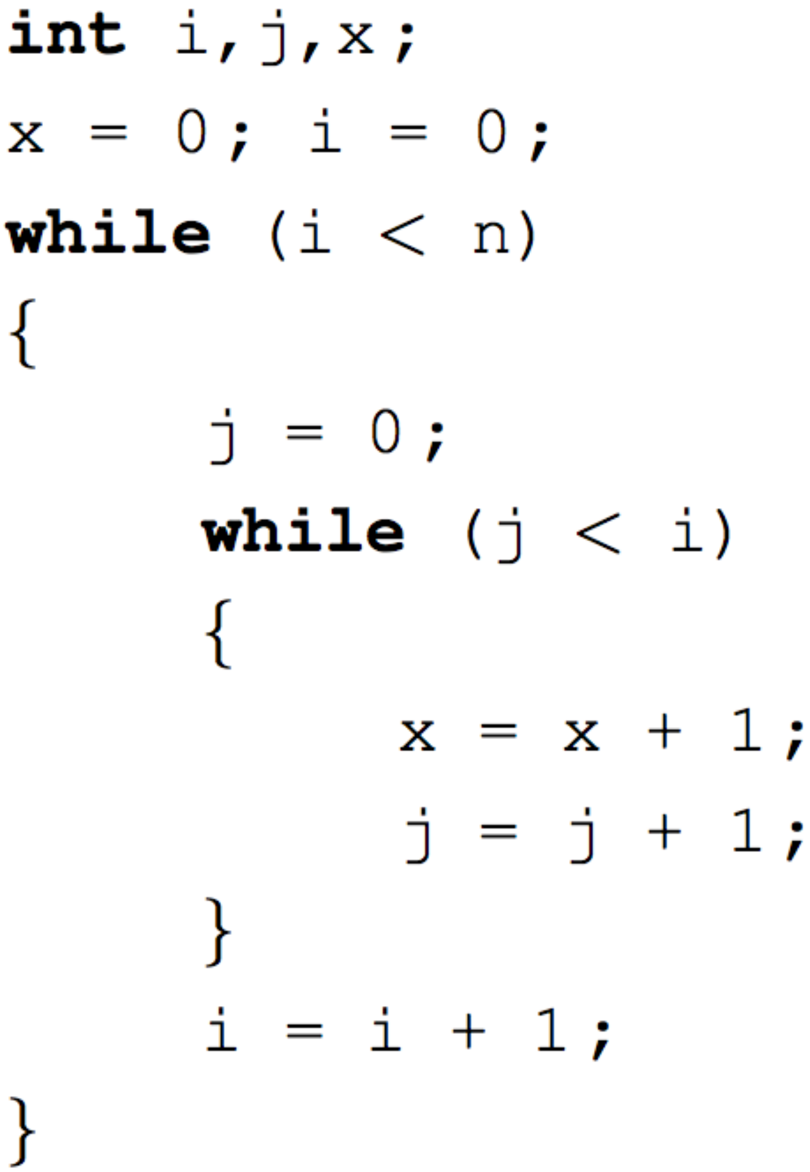
Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < i)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Juste une petite modification :
le test
(j < n)
est remplacé par
(j < i)

Incidence sur la complexité ?

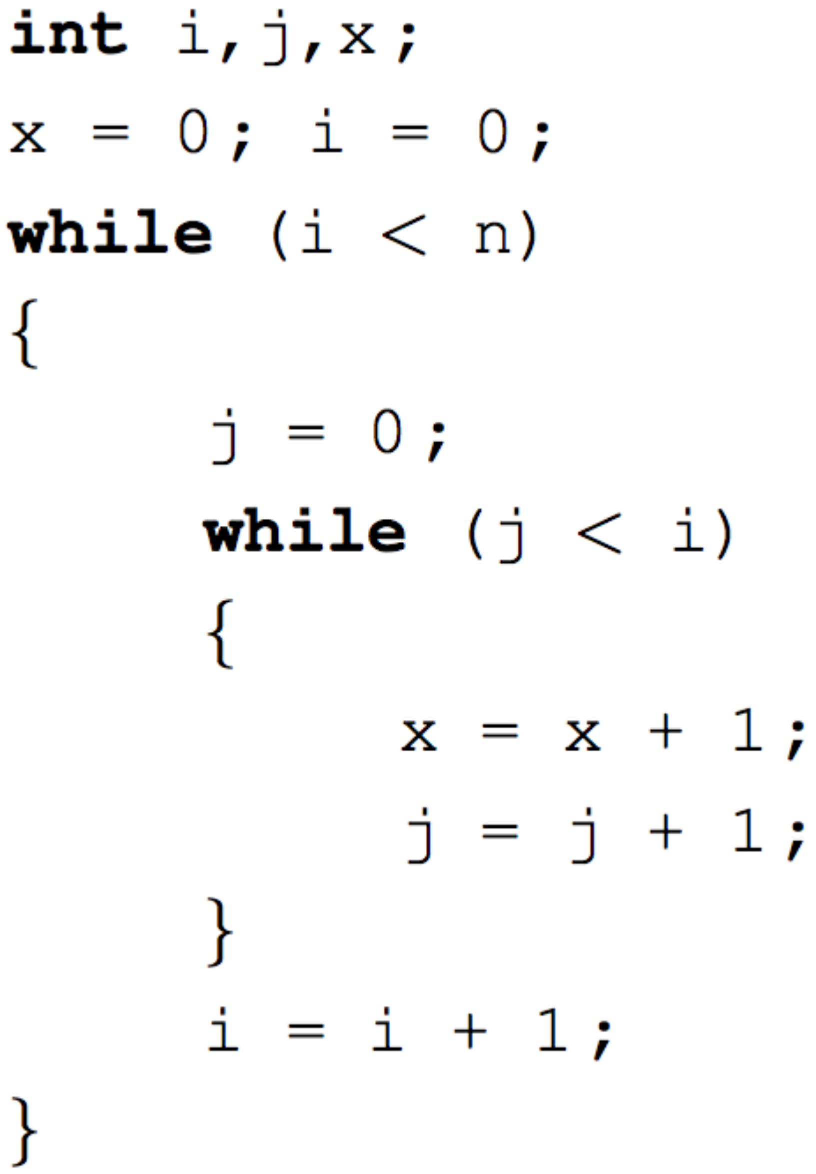
Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse :

Algorithme A_5 : une variation sur A_4

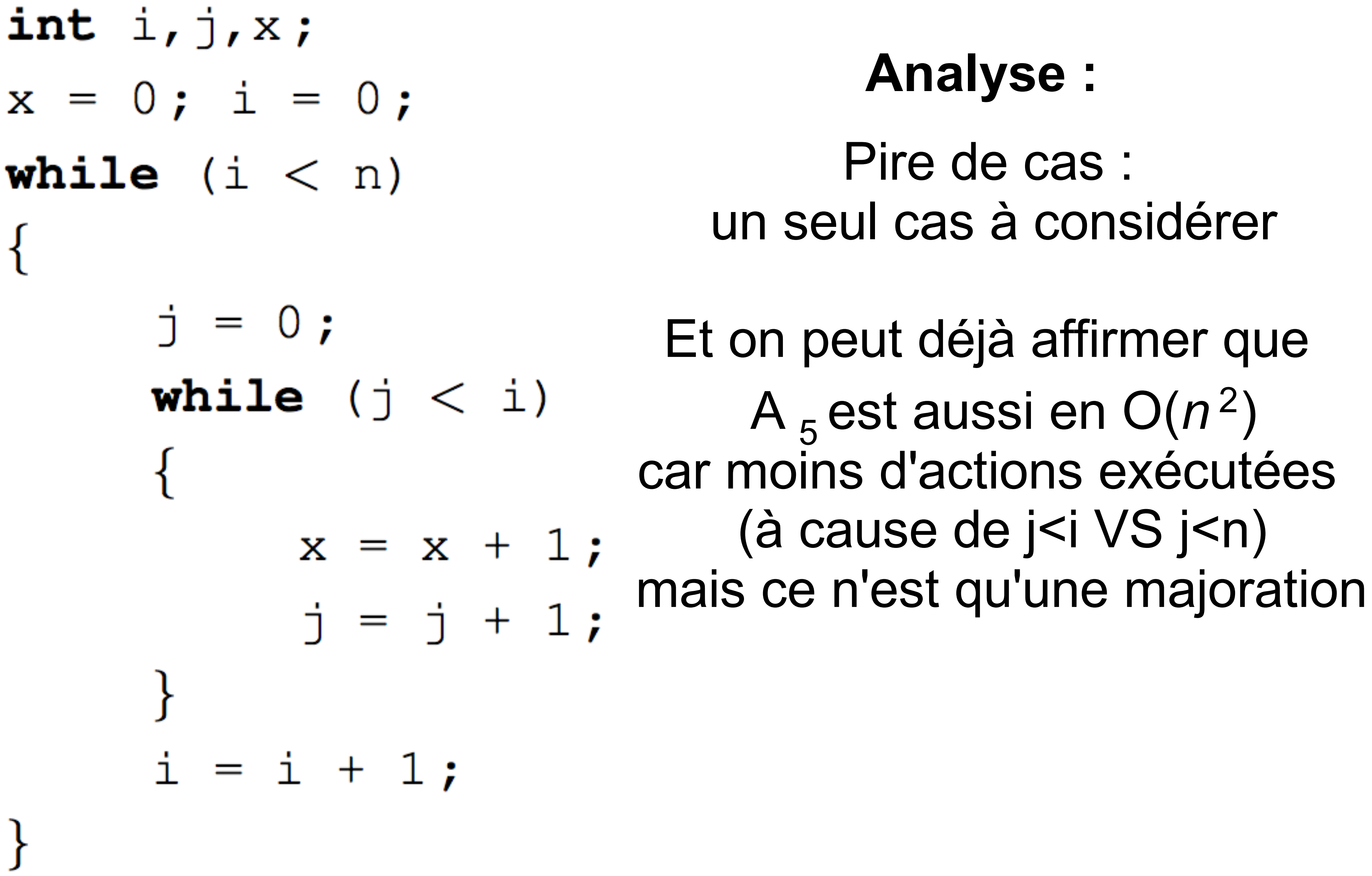


```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse :

Pire de cas :
un seul cas à considérer

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
     j = 0;
     while (j < i)
     {
          x = x + 1;
          j = j + 1;
     }
     i = i + 1;
}
```

Analyse :

Pire de cas :
un seul cas à considérer

Et on peut déjà affirmer que
A $_5$ est aussi en O(n^2)
car moins d'actions exécutées
(à cause de j<i VS j<n)
mais ce n'est qu'une majoration

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
     j = 0;
     while (j < i)
     {
          x = x + 1;
          j = j + 1;
     }
     i = i + 1;
}
```

Analyse :

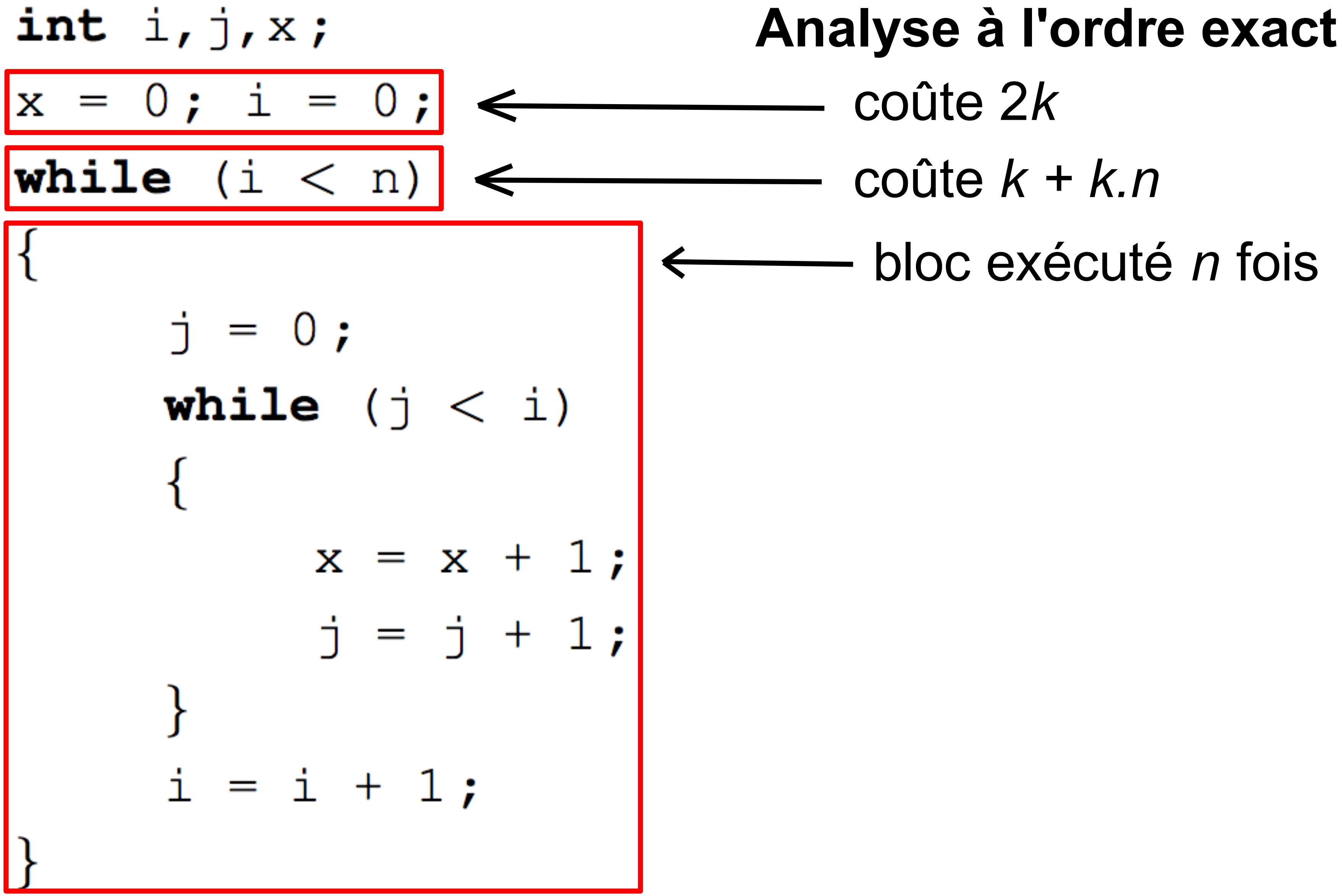
Pire de cas :
un seul cas à considérer

Et on peut déjà affirmer que
A_5 est aussi en $O(n^2)$
car moins d'actions exécutées
(à cause de j<i VS j<n)
mais ce n'est qu'une majoration

Quid de l'ordre exact ?
(cf. notation théta)

Algorithme A_5 : une variation sur A_4

Analyse à l'ordre exact



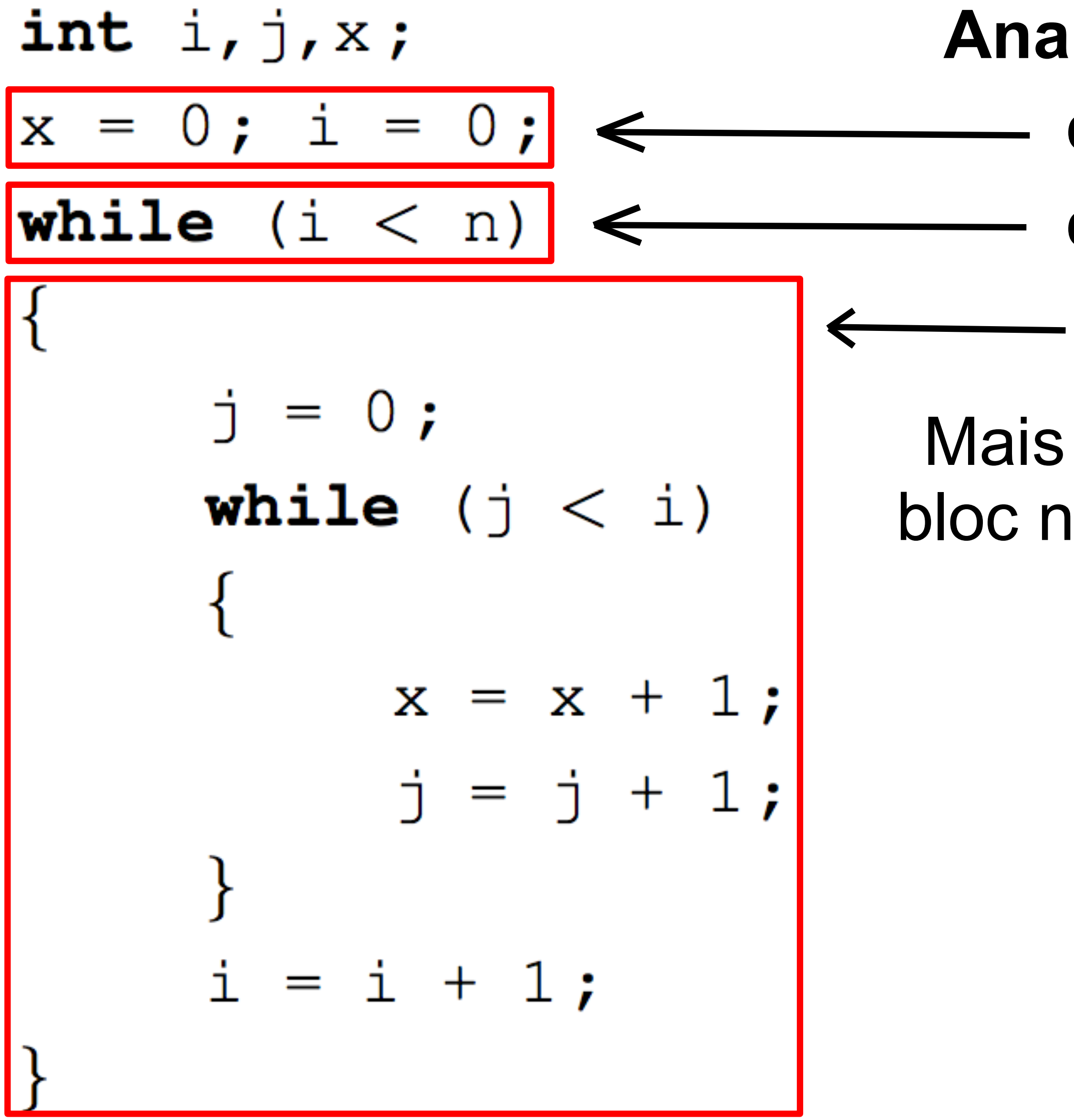
```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

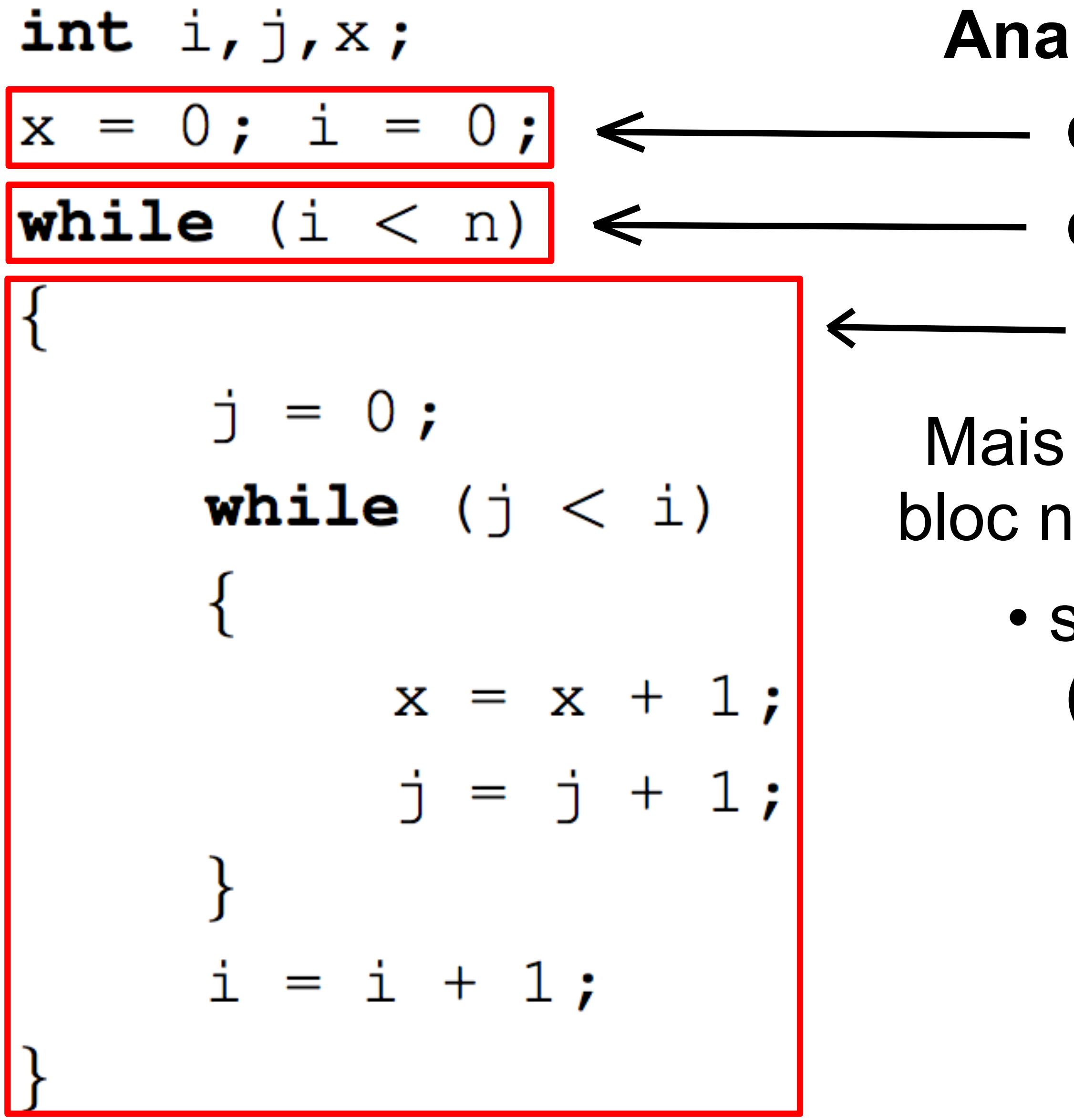
coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Mais chaque exécution du bloc n'a pas le même coût :

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

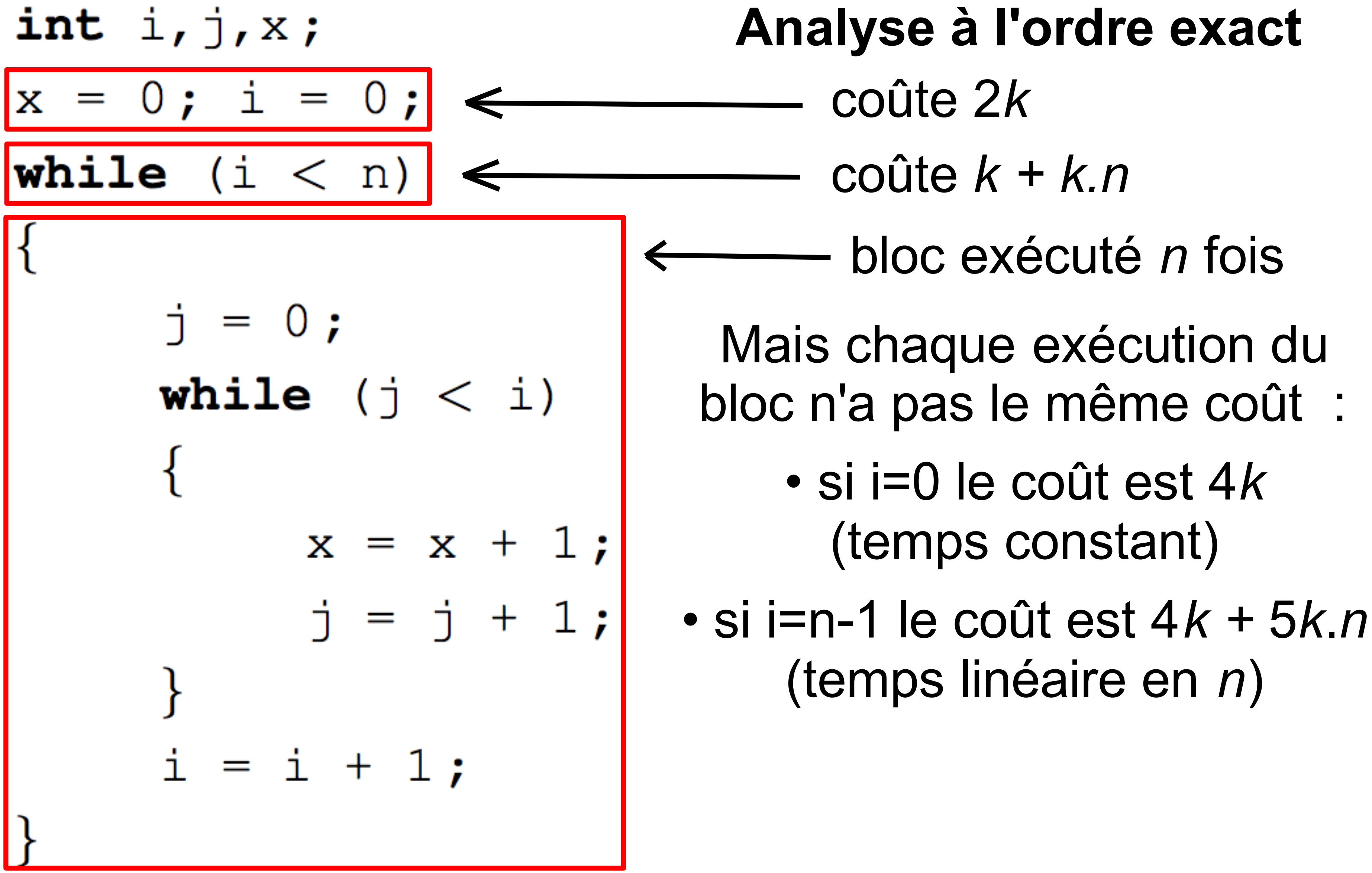
coûte $k + k.n$

bloc exécuté n fois

Mais chaque exécution du bloc n'a pas le même coût :

- si i=0 le coût est $4k$ (temps constant)

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

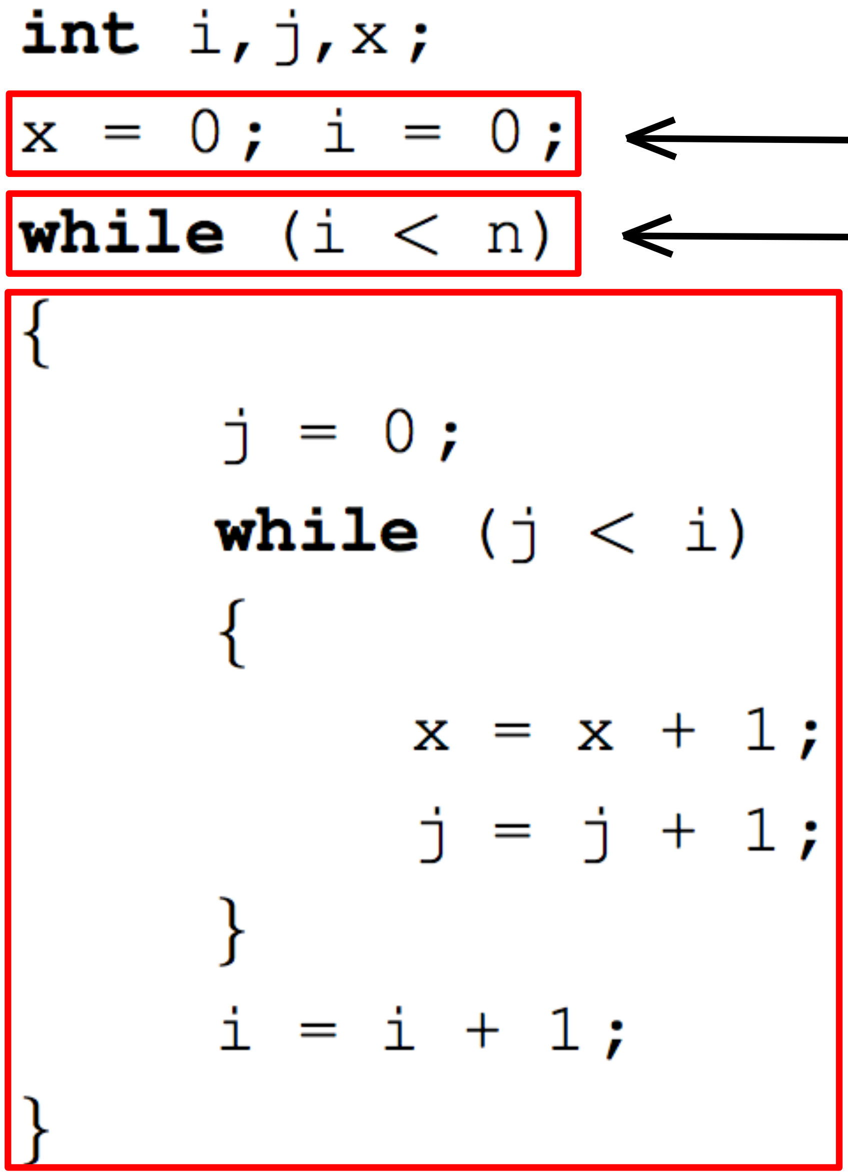
coûte $k + k.n$

bloc exécuté *n* fois

Mais chaque exécution du bloc n'a pas le même coût :

- si i=0 le coût est $4k$ (temps constant)
- si i=n-1 le coût est $4k + 5k.n$ (temps linéaire en *n*)

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

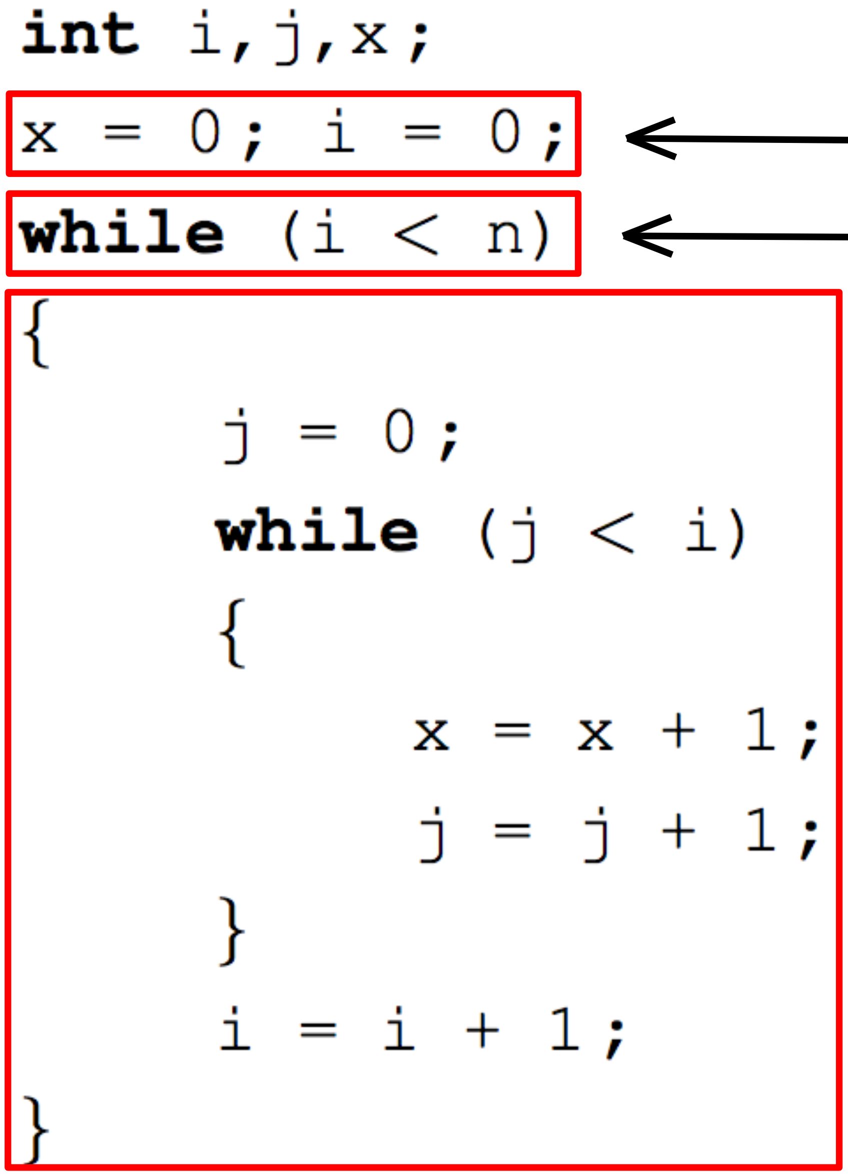
bloc exécuté n fois

Mais chaque exécution du bloc n'a pas le même coût :

- si i=0 le coût est $4k$ (temps constant)
- si i=n-1 le coût est $4k + 5k.n$ (temps linéaire en n)

L'analyse est plus subtile !

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < i)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Analyse à l'ordre exact

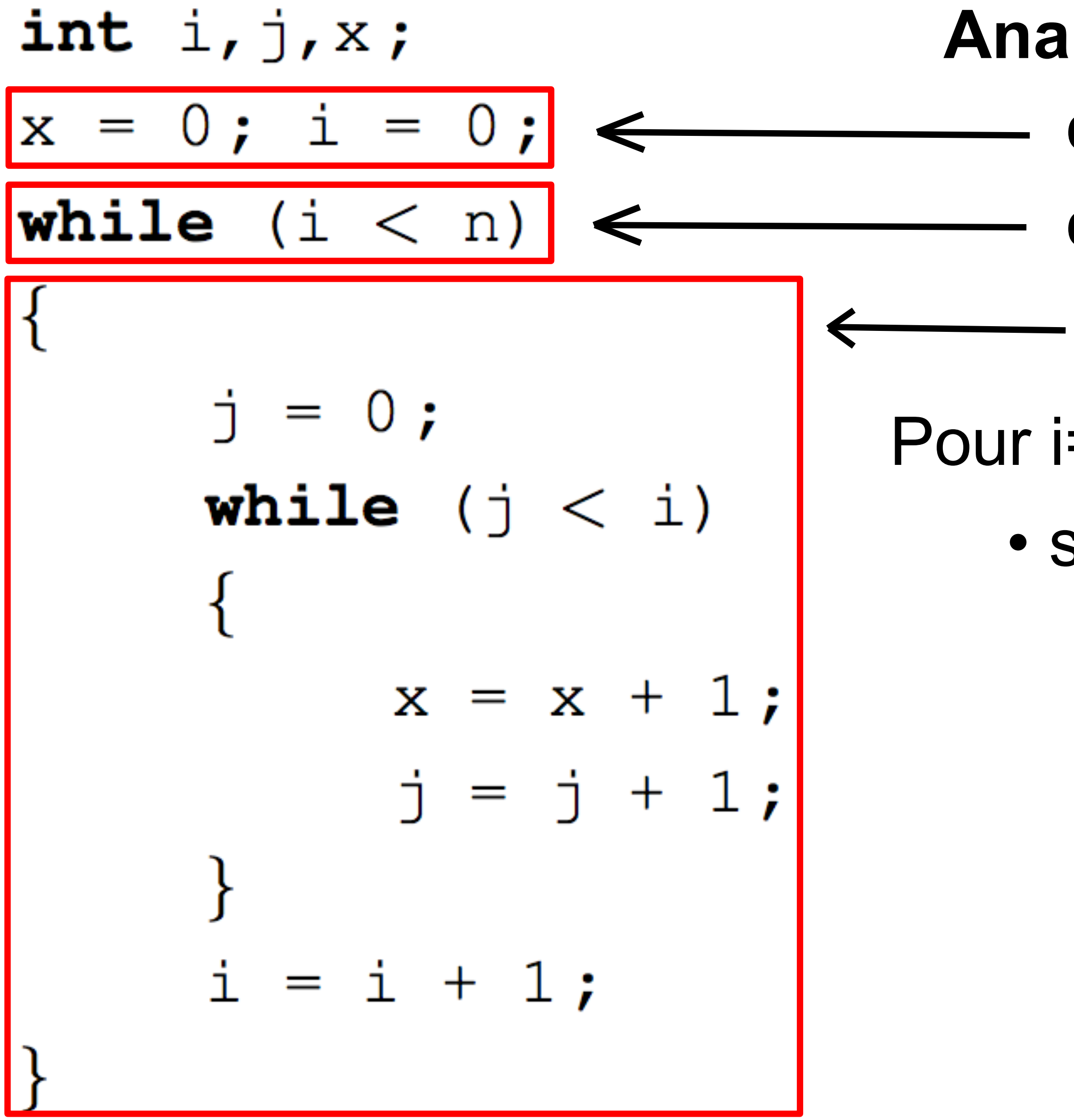
coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

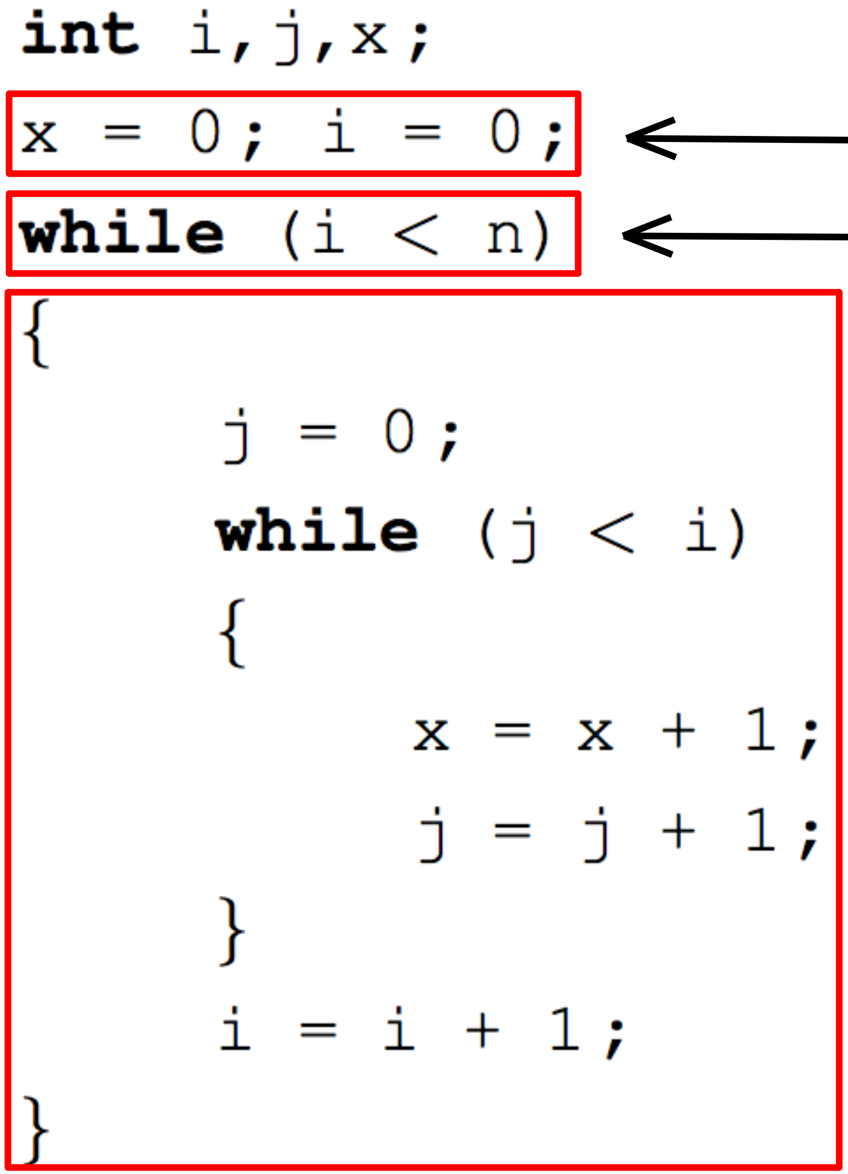
coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
     j = 0;
     while (j < i)
     {
          x = x + 1;
          j = j + 1;
     }
     i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

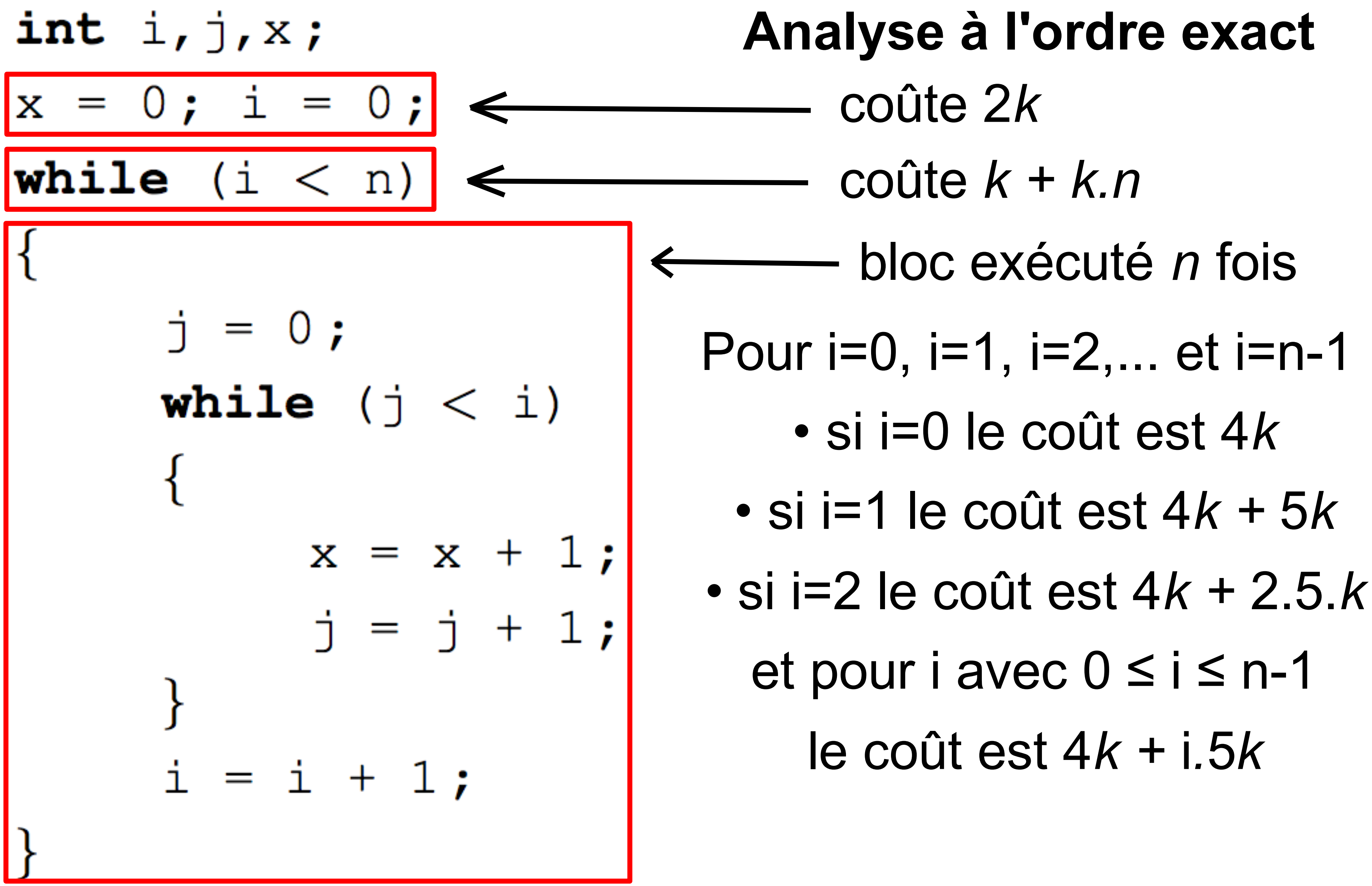
coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$
- si i=2 le coût est $4k + 2.5.k$

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

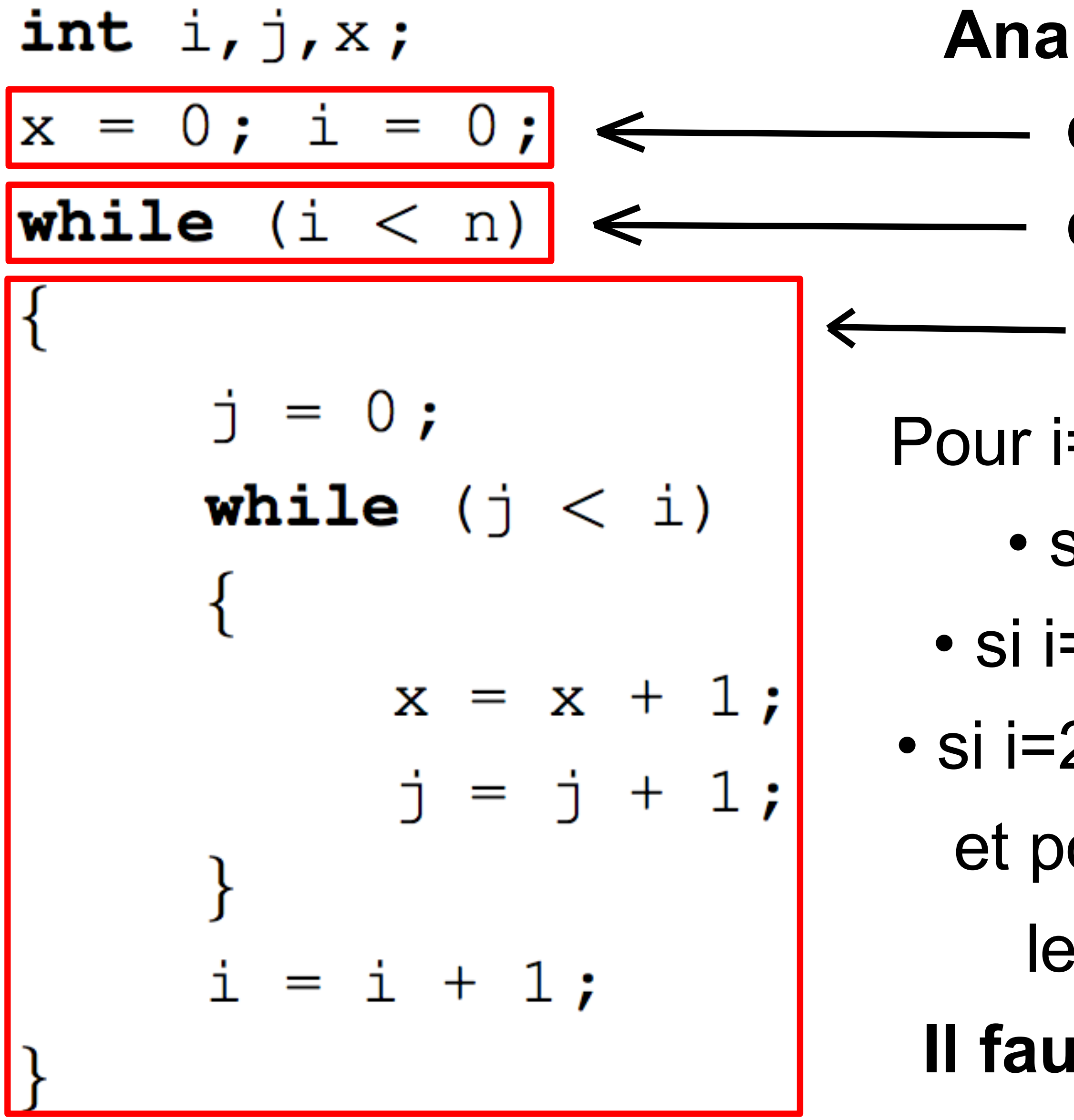
Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$
- si i=2 le coût est $4k + 2.5.k$

et pour i avec $0 \leq i \leq n-1$

le coût est $4k + i.5k$

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

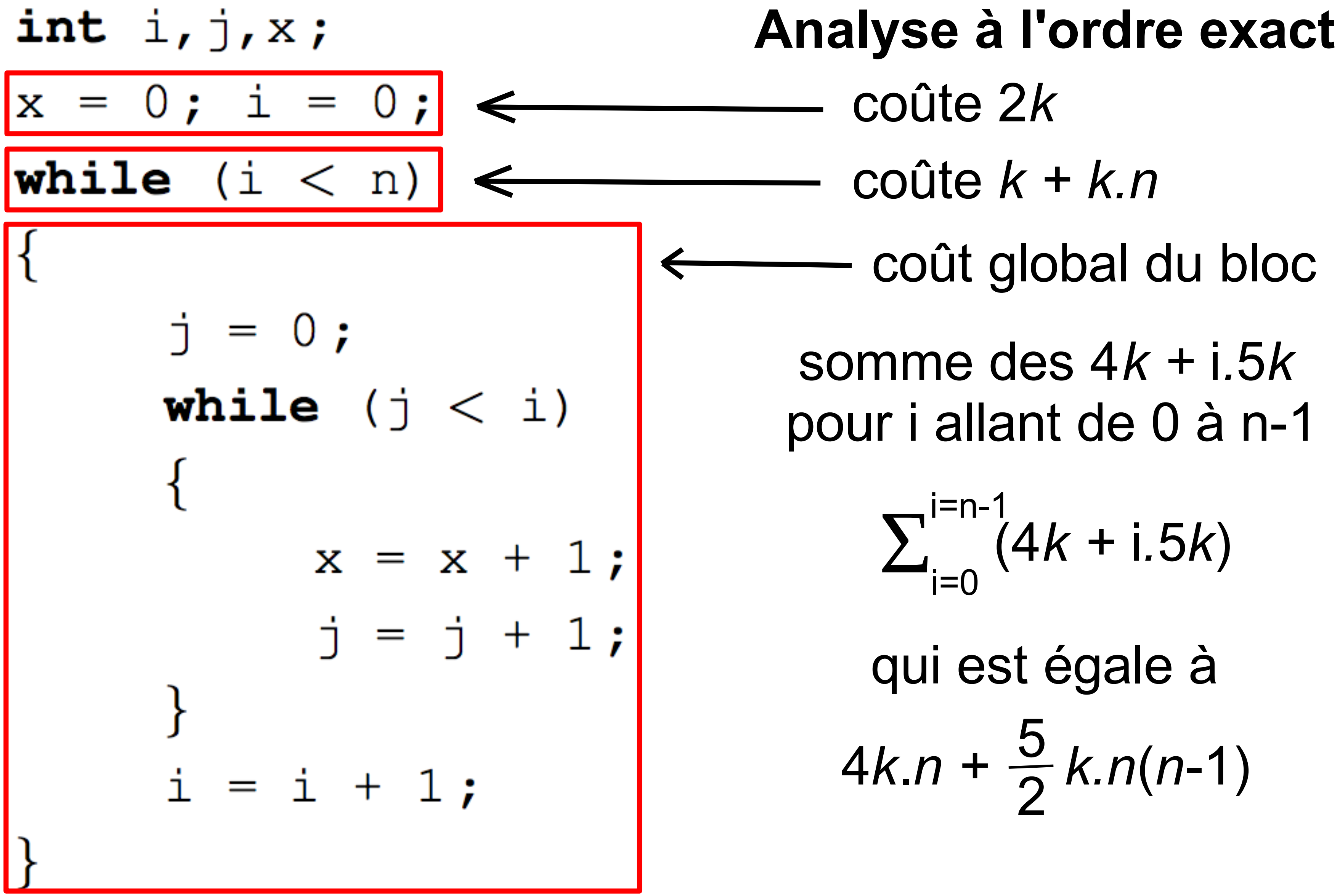
- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$
- si i=2 le coût est $4k + 2.5.k$

et pour i avec $0 \leq i \leq n-1$

le coût est $4k + i.5k$

Il faut donc additionner !

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

coût global du bloc

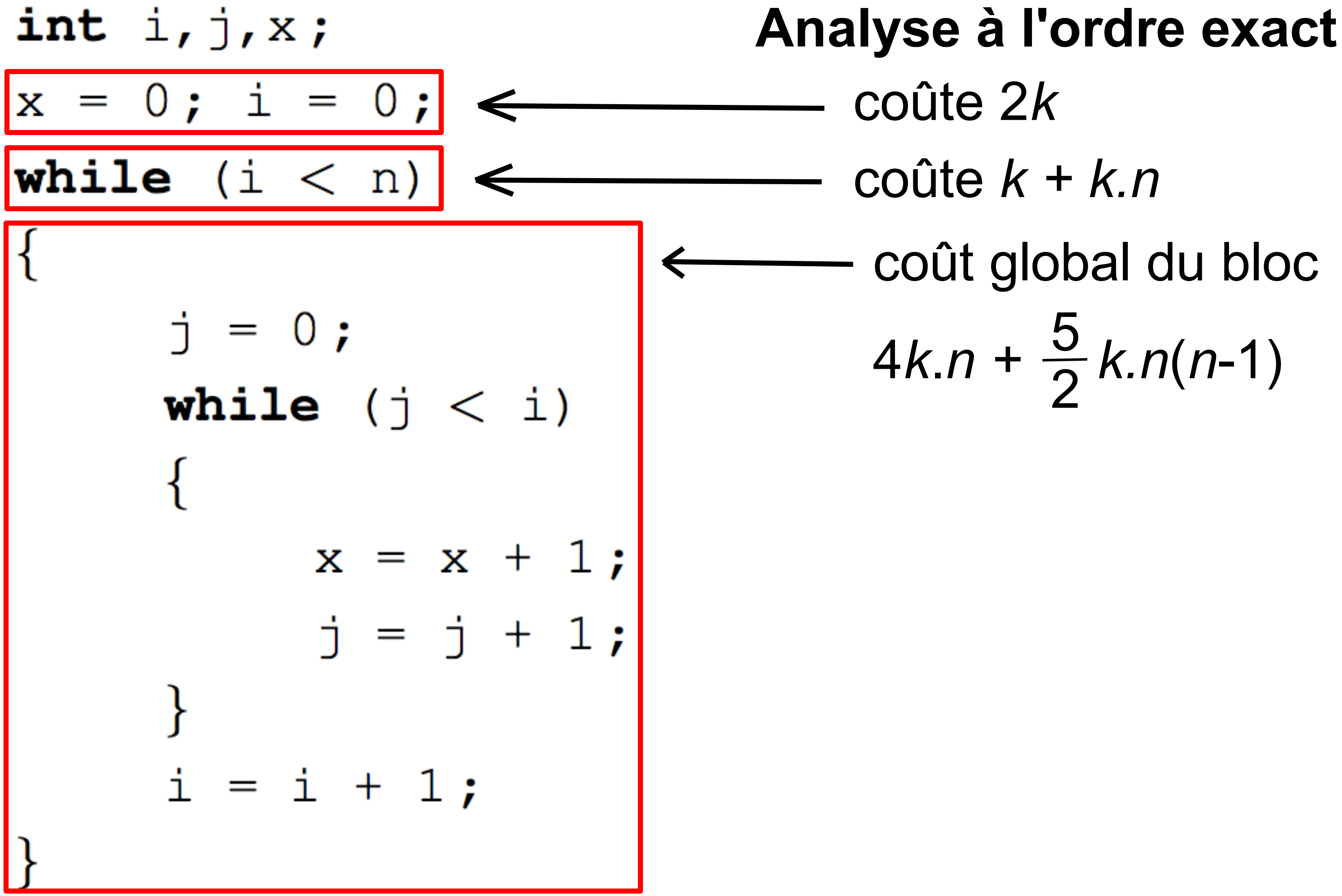
somme des $4k$ + i.$5k$
pour i allant de 0 à n-1

$$\sum_{i=0}^{i=n-1}(4k + i.5k)$$

qui est égale à

$$4k.n + \frac{5}{2}k.n(n-1)$$

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < i)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Analyse à l'ordre exact

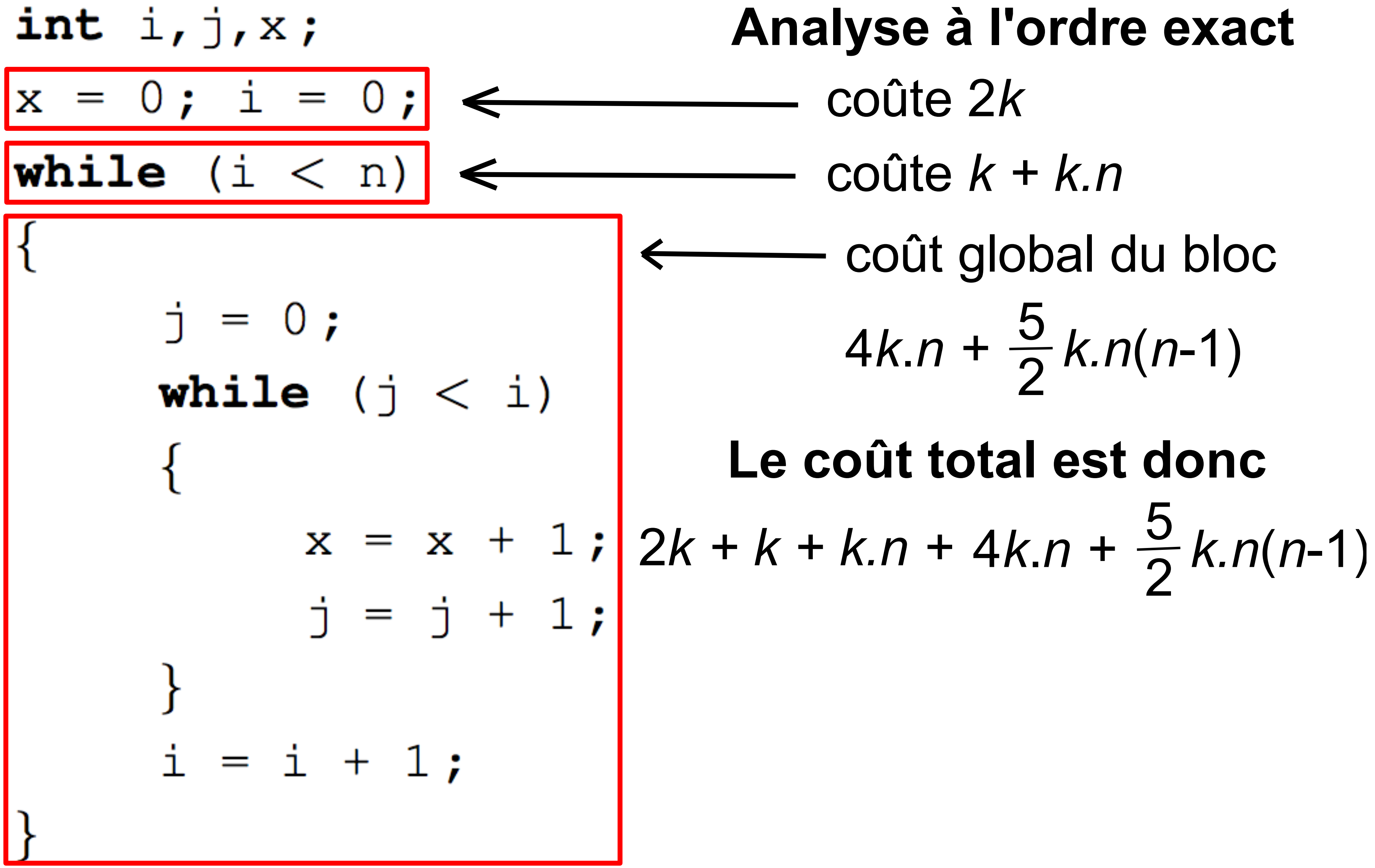
coûte $2k$

coûte $k + k.n$

coût global du bloc

$$4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

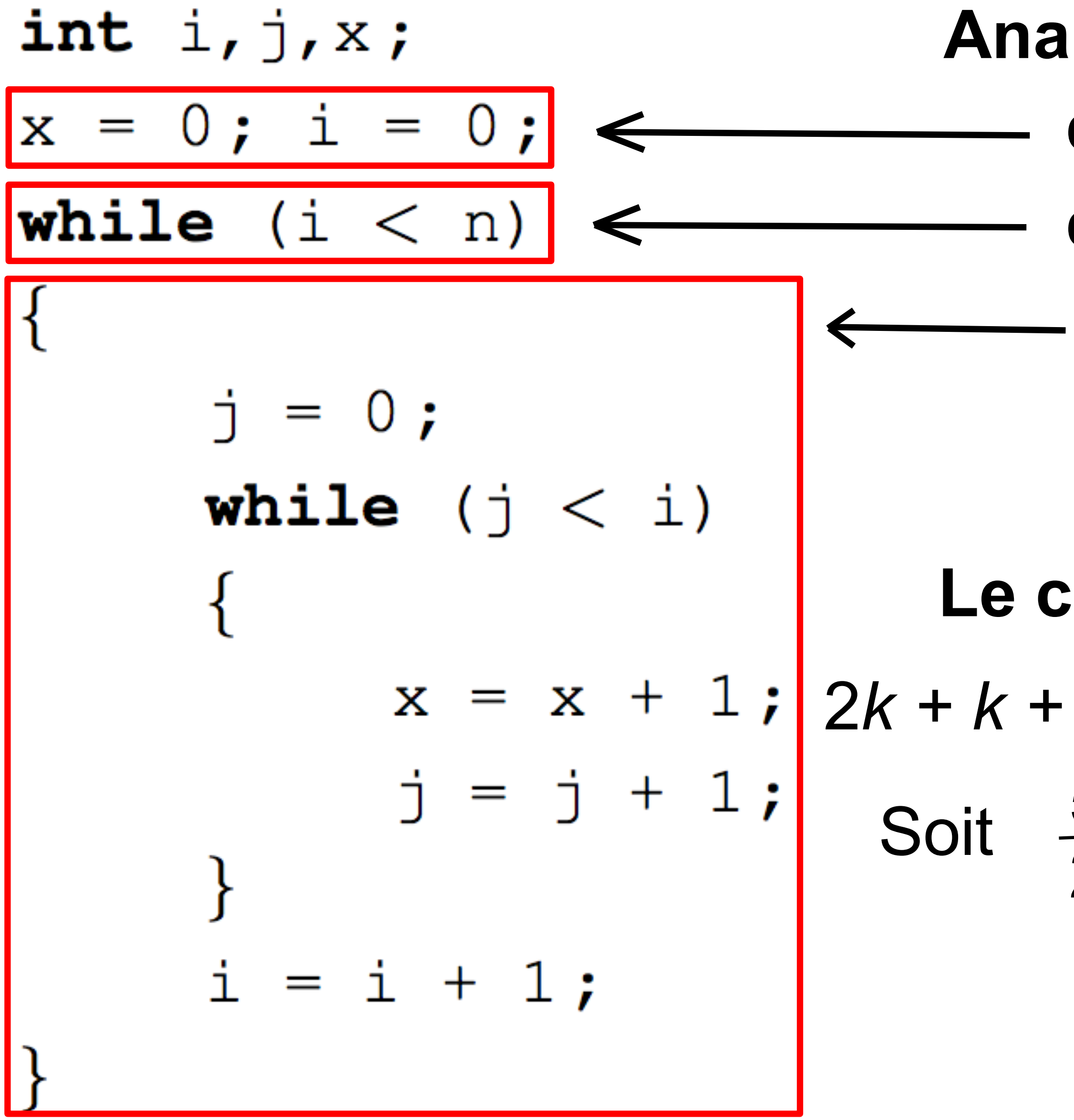
coût global du bloc

$$4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Le coût total est donc

$$2k + k + k.n + 4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Algorithme A_5 : une variation sur A_4



```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

coût global du bloc

$$4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Le coût total est donc

$$2k + k + k.n + 4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Soit $\frac{5}{2}k.n^2 + \frac{5}{2}k.n + 3k$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

coût global du bloc

$$4k.n + \frac{5}{2}k.n(n-1)$$

Le coût total est donc

$$2k + k + k.n + 4k.n + \frac{5}{2}k.n(n-1)$$

Soit $\frac{5}{2}k.n^2 + \frac{5}{2}k.n + 3k$

A_5 est donc en $\Theta(n^2)$

Plan

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)
2. Graphes et Algorithmes : des rappels
   - Notions de base
     - Présentation
     - Définitions et terminologie
     - Représentation machine
   - Chemins, Parcours et ConnexitéS
     - Chemins, chaînes, parcours et numérotations
     - Connexité et forte connexité
3. Rappels : pour conclure

Majoration : notation "Grand O"

La complexité est toujours évaluée en ordre de grandeur asymptotique :

Majoration : notation "Grand O"

La complexité est toujours évaluée en ordre de grandeur asymptotique :

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

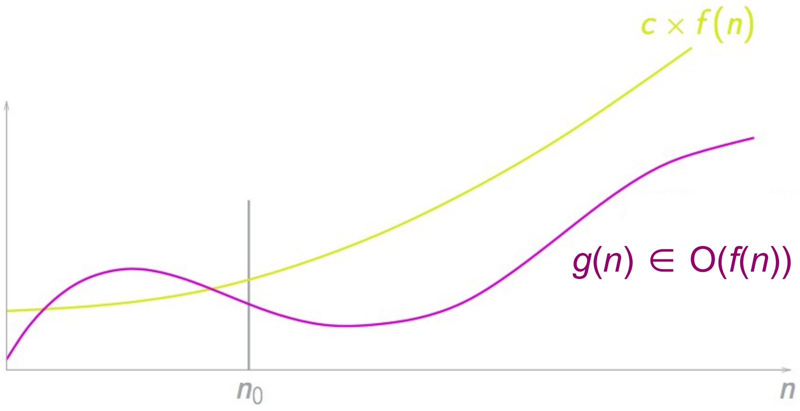
$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$

(merci à Stéphane Grandcolas pour le tracé des courbes)

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$



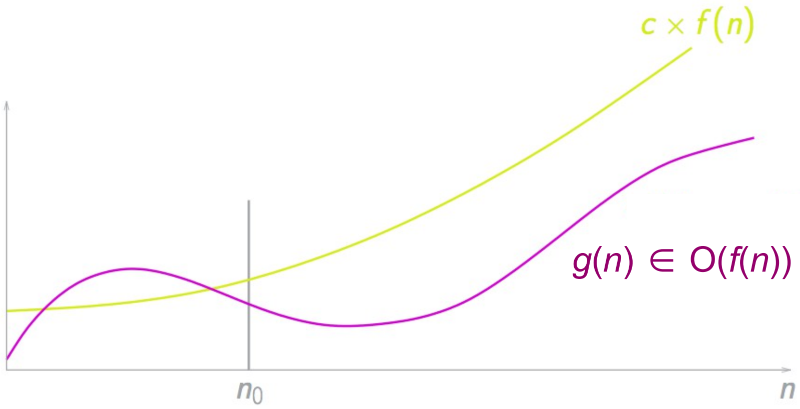
Pourquoi peut-on écrire que $g(n) = 100n + 400 \in O(n)$ (avec $f(n) = n$) ?

- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$



Pourquoi peut-on écrire que $g(n) = 100n + 400 \in O(n)$ (avec $f(n) = n$) ?

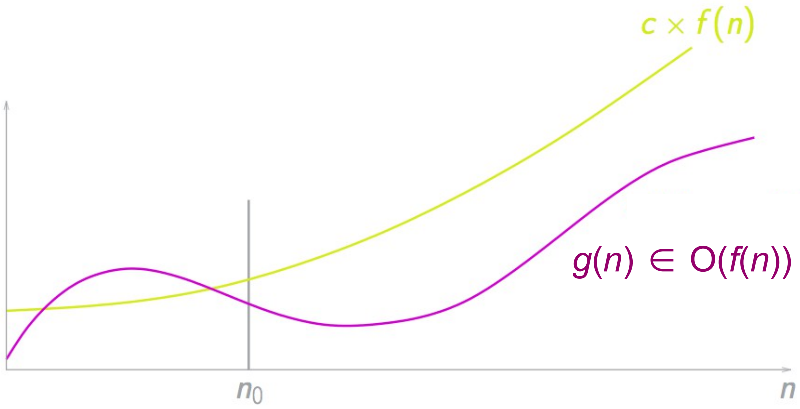
- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)
- parce que avec $c = 300$ et $n_0 = 2$, $\forall n \geq n_0$, on a $g(n) \leq c.f(n)$

  en effet $g(1) = 500$ et $c.f(1) = 300$,

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$



Pourquoi peut-on écrire que $g(n) = 100n + 400 \in O(n)$ (avec $f(n) = n$) ?

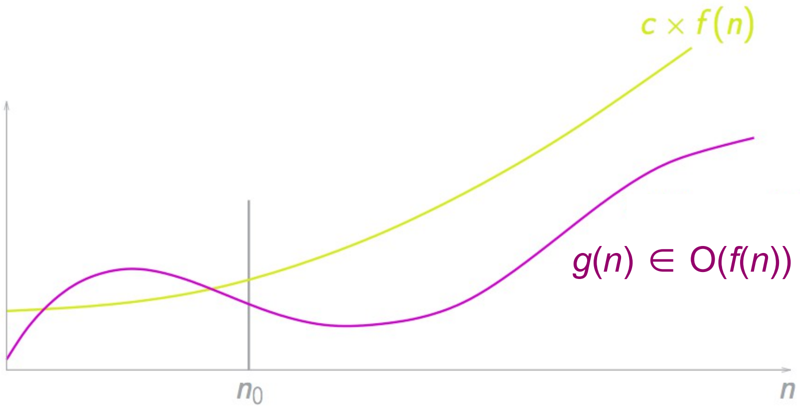
- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)
- parce que avec $c = 300$ et $n_0 = 2$, $\forall n \geq n_0$, on a $g(n) \leq c.f(n)$

  en effet $g(1) = 500$ et $c.f(1) = 300$, puis $g(2) = 600 = c.f(2)$,

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \to \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$



Pourquoi peut-on écrire que $g(n) = 100n + 400 \in O(n)$ (avec $f(n) = n$) ?

- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)
- parce que avec $c = 300$ et $n_0 = 2$, $\forall n \geq n_0$, on a $g(n) \leq c.f(n)$

  en effet $g(1) = 500$ et $c.f(1) = 300$,   puis $g(2) = 600 = c.f(2)$,

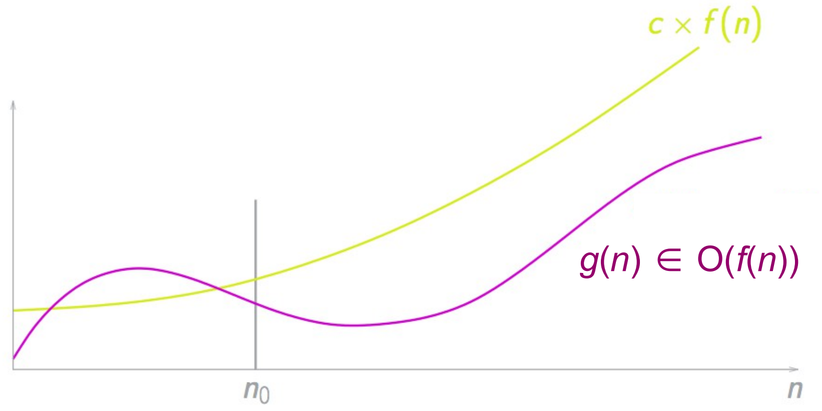
  puis $g(3) = 700$ et $c.f(3) = 900$,   puis $g(4) = 800$ et $c.f(4) = 1200$, etc.

Majoration : notation "Grand O"

Commentaires :

- La fonction $g(n)$ exprime ici un temps d'exécution $T_A(n)$
- Permet de caractériser facilement des complexités :
  - temps constant : en $O(1)$
  - temps logarithmique : en $O(\log(n))$
  - temps linéaire : en $O(n)$
  - temps linéarithmique : en $O(n.\log(n))$
  - temps quadratique : en $O(n^2)$
  - temps cubique : en $O(n^3)$
  - temps exponentiel : en $O(2^n)$
  - etc.

Majoration : notation "Grand O"

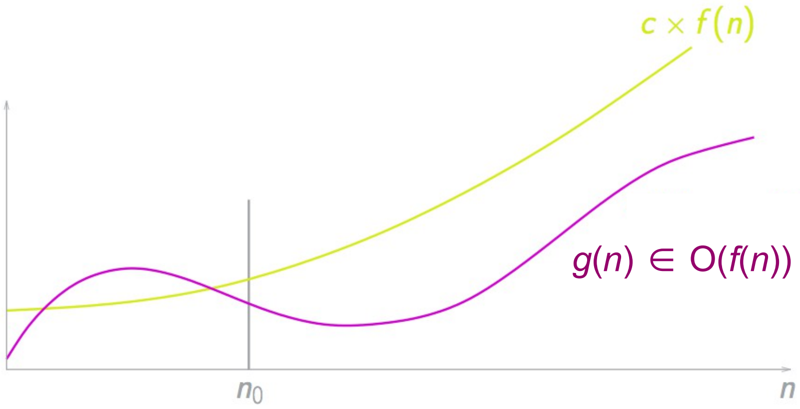


Commentaires :

- Permet de se concentrer sur l'essentiel : la nature du comportement
  - supprime les termes négligeables : n^2 est "oublié" face à n^3
  - élimine les facteurs multiplicatifs constants : $437/5.n^3$ s'écrira n^3

en effet : $g(n) = 2542.n^3 + 10^5.n^2 + 47/9.n \in O(n^3)$

Majoration : notation "Grand O"



Commentaires :

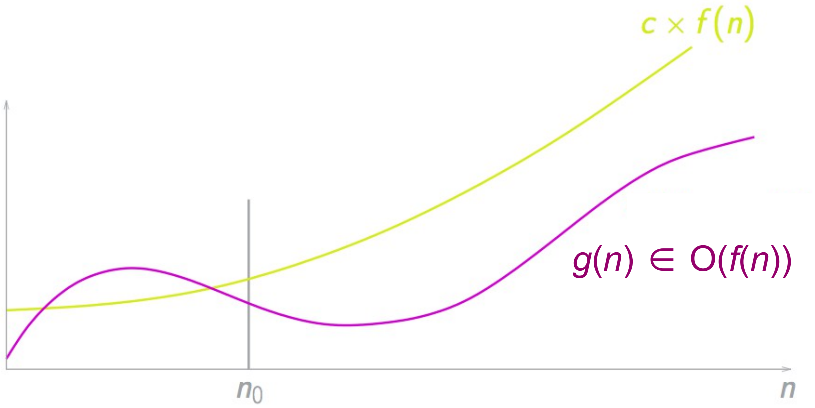
- Permet de se concentrer sur l'essentiel : la nature du comportement
  - supprime les termes négligeables : n^2 est "oublié" face à n^3
  - élimine les facteurs multiplicatifs constants : $437/5.n^3$ s'écrira n^3

en effet : $g(n) = 2542.n^3 + 10^5.n^2 + 47/9.n \in O(n^3)$

- Parfois, il est écrit "$g(n) = n^2 + n \mathbf{=} O(n^2)$"
  - mais selon D. Knuth (1976), O exprime des ensembles de fonctions...
  - il semble donc souhaitable d'écrire "$g(n) \in O(n^2)$"

(on dit souvent "$g(n)$ **est en** $O(n^2)$" par traduction de "**is in**")

Majoration : notation "Grand O"



Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct !

  toute fonction linéaire est majorée asymptotiquement par une fonction quadratique

Majoration : notation "Grand O"

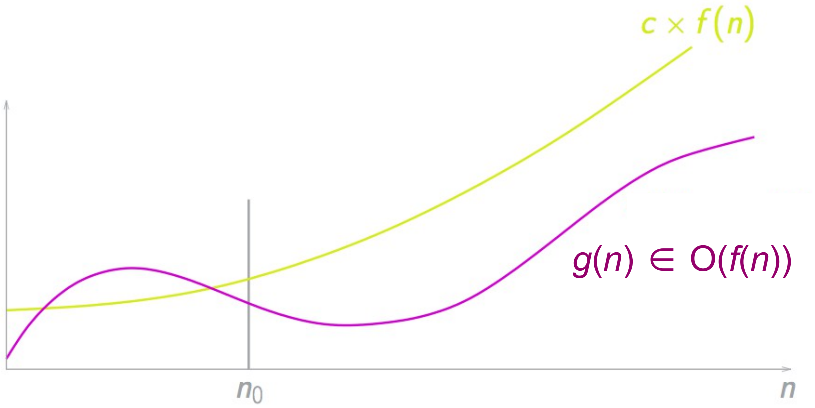
Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct !
  toute fonction linéaire est majorée asymptotiquement par une fonction quadratique
- Utilisation de la notation "O" : c'est imprécis
  - l'ordre exact n'est pas exprimé
  - une forme d'aveu d'impuissance, d'incompétence

  O n'est à n'utiliser que si l'ordre exact est inconnu ou quand c'est suffisant

  (comme dans ce cours en général...)

Majoration : notation "Grand O"



Commentaires :

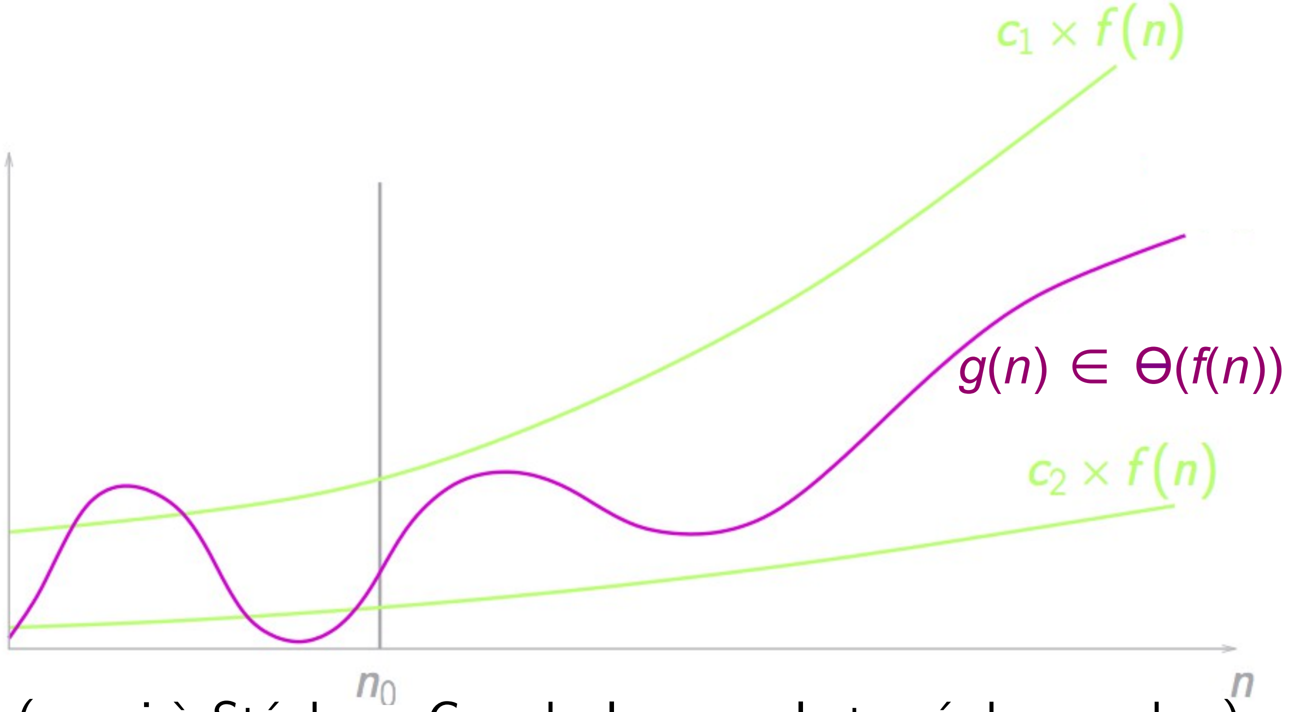
- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct !
  toute fonction linéaire est majorée asymptotiquement par une fonction quadratique
- Utilisation de la notation "O" : c'est imprécis
  - l'ordre exact n'est pas exprimé
  - une forme d'aveu d'impuissance, d'incompétence

  O n'est à n'utiliser que si l'ordre exact est inconnu ou quand c'est suffisant (comme dans ce cours en général...)
- À ne pas confondre avec la notation "petit o" : $g(n) \in o(f(n))$
  (si g est négligeable devant f asymptotiquement : $\forall c \in \mathbb{R}^+, \ldots g(n) \leq c.f(n)$)

Ordre exact : notation "Théta"

Ordre exact (notation "Θ", dire "Théta"): Etant donnée une fonction $f : \mathbb{N} \to R^*$, l'ensemble des fonctions bornées supérieurement et inférieurement par des multiples réels de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$\Theta(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c_1, c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, c_2.f(n) \leq g(n) \leq c_1.f(n)\}$$



(merci à Stéphane Grandcolas pour le tracé des courbes)

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)
- Majoration ("Grand O") :
  - quand c'est suffisant (comme dans ce cours en général))
  - ou quand on ne sait pas faire mieux

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)
- Majoration ("Grand O") :
  - quand c'est suffisant (comme dans ce cours en général))
  - ou quand on ne sait pas faire mieux
- La minoration existe : "Grand Omega" avec la notation "Ω" :
  - $\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \geq c.f(n)\}$

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)
- Majoration ("Grand O") :
  - quand c'est suffisant (comme dans ce cours en général))
  - ou quand on ne sait pas faire mieux
- La minoration existe : "Grand Omega" avec la notation "Ω" :
  - $\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \geq c.f(n)\}$
  - peut être utile avec la propriété : $\Theta((f(n)) = O(f(n)) \cap \Omega(f(n))$ si exprimer Θ() n'est pas immédiat et que O et Ω sont évidents

Plan

Incidence du codage

Complexité d'un algorithme : toujours fonction de la taille des entrées

$\Rightarrow$ il faut alors considérer des "**codages raisonnables**" :

Incidence du codage

Complexité d'un algorithme : toujours fonction de la taille des entrées

$\Rightarrow$ il faut alors considérer des "**codages raisonnables**" :

Exemple du traitement d'un entier n :

- Plusieurs codages sont *a priori* possibles dont :
  - codage unaire : n batonnets (souvenirs de l'école maternelle...)
  - codage binaire : $\lfloor log_2(n) \rfloor + 1$ bits

Incidence du codage

Complexité d'un algorithme : toujours fonction de la taille des entrées

$\Rightarrow$ il faut alors considérer des "**codages raisonnables**" :

Exemple du traitement d'un entier n :

- Plusieurs codages sont *a priori* possibles dont :
  - codage unaire : n batonnets (souvenirs de l'école maternelle...)
  - codage binaire : $\lfloor log_2(n) \rfloor + 1$ bits
- Le choix du codage a alors une incidence considérable !

  Un algorithme peut "devenir" polynomial avec un "mauvais codage".

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

```
Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{\text{oui,non}\}$
  $prem \leftarrow$ oui
  $i \leftarrow 2$
  tantque $(prem$ AND $(i \leq \sqrt{n}))$ faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
  fin tantque
  Retourner $prem$.
```

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

```
Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{oui,non\}$
  $prem \leftarrow$ oui
  $i \leftarrow 2$
  tantque ($prem$ AND ($i \leq \sqrt{n}$)) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
  fin tantque
  Retourner $prem$.
```

Complexité :

- au plus $\sqrt{n} - 1$ passages dans la boucle **tantque**

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

```
Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{oui,non\}$
  $prem \leftarrow$ oui
  $i \leftarrow 2$
  tantque ($prem$ AND ($i \leq \sqrt{n}$)) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
  fin tantque
  Retourner $prem$.
```

Complexité :

- au plus $\sqrt{n} - 1$ passages dans la boucle **tantque**
- supposition : traitement local à la boucle réalisable en temps constant

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

```
Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{\text{oui,non}\}$
  $prem \leftarrow$ oui
  $i \leftarrow 2$
  tantque ($prem$ AND ($i \leq \sqrt{n}$)) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
  fin tantque
  Retourner $prem$.
```

Complexité :

- au plus $\sqrt{n} - 1$ passages dans la boucle **tantque**
- supposition : traitement local à la boucle réalisable en temps constant

La complexité est donc $\Theta(\sqrt{n}) = \Theta(n^{1/2})$, soit mieux que linéaire ?

Incidence du codage

Cet algorithme très simple

```
$prem \leftarrow$ oui
$i \leftarrow 2$
**tantque** $(prem$ AND $(i \leq \sqrt{n}))$ **faire**
    **si** ( $i$ divise $n$ ) **alors** $prem \leftarrow$ non
    **sinon** $i \leftarrow i + 1$
**fin tantque**
```

Incidence du codage

Cet algorithme très simple

```
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND ($i \leq \sqrt{n}$)) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
fin tantque
```

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

Incidence du codage

Cet algorithme très simple

```
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND ($i \leq \sqrt{n}$)) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
fin tantque
```

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- ”racinaire” : mieux que **polynomial** pour le codage unaire

Incidence du codage

Cet algorithme très simple

```
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND $(i \leq \sqrt{n})$) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
fin tantque
```

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- "racinaire" : mieux que **polynomial** pour le codage unaire
- (sous-)**exponentielle pour le codage binaire !**

Incidence du codage

Cet algorithme très simple

```
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND ($i \leq \sqrt{n}$)) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
fin tantque
```

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- "racinaire" : mieux que **polynomial** pour le codage unaire
- (sous-)**exponentielle pour le codage binaire !**
  - taille de la donnée : $Taille(n) = \lfloor log_2(n) \rfloor + 1$
  - d'où $n \in \Theta(2^{Taille(n)-1})$
  - complexité de l'algorithme : $\Theta(2^{[Taille(n)-1]/2})$

    qui est (sous-)exponentielle dans la taille de la donnée

Incidence du codage

Cet algorithme très simple

```
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND ($i \leq \sqrt{n}$)) faire
    si ( $i$ divise $n$ ) alors $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
fin tantque
```

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- "racinaire" : mieux que **polynomial** pour le codage unaire
- (sous-)**exponentielle pour le codage binaire !**
  - taille de la donnée : $Taille(n) = \lfloor log_2(n) \rfloor + 1$
  - d'où $n \in \Theta(2^{Taille(n)-1})$
  - complexité de l'algorithme : $\Theta(2^{[Taille(n)-1]/2})$
    qui est (sous-)exponentielle dans la taille de la donnée

On travaillera toujours avec des **codages raisonnables**