Algorithmique Avancée

Problèmes Récursifs

Présentation

- Enseignant :

  M. Abdessamad IMINE

  Bureau : 140

  email : Abdessamad.Imine@univ-lorraine.fr

- Progression du module :

  Cours, TD et TP

- Examens :

  Un partiel (fin octobre)

  Un projet à rendre

Introduction

- Une procédure est dite récursive si, et seulement si, elle fait appel à elle-même, soit directement soit indirectement
- La récursivité est une technique de résolution de problème utilisée en informatique.
- Elle permet d'écrire des solutions **courtes** pour des problèmes complexes : on réduit le problème initial à un problème similaire mais de taille plus petite.
- Elle permet de trouver des solutions **élégantes** et **claires** à certains problèmes

Exemple

-(vision itérative)
Un escalier de hauteur h c'est : **une séquence de h marches**

- (vision récursive)
Un escalier de hauteur h c'est : **une marche suivie d'un escalier de hauteur h − 1**

Exemple (suite)

Version itérative :

```
fonction monter_escalier( h : entier )
début
pour  i de 1 à h faire
     monter_marche()
fpour
fin
```

Version récursive :

```
fonction monter_escalier( h : entier )
début
si h>0
alors      monter_marche()
           monter_escalier(h-1)
fsi
fin
```

Exemple (suite)

Récursivité en action

• Que fait l'appel monter_escalier(3) ?

```
 monter_escalier( 3 )
=
monter_marche();
monter_escalier( 2 );
 =
monter_marche();
monter_marche();
monter_escalier( 1 );
=
monter_marche();
monter_marche();
monter_marche();
```

• Même effet que la version itérative, c'est-à-dire 3 appels à monter_marche()

Condition d’arrêt

• Tout algorithme récursif comporte une instruction (ou un bloc d'instructions) nommée condition d’arrêt, qui garantit la fin des appels récursifs et qui porte sur les paramètres.

• Pour tout problème récursif, il faut :
  • préciser la condition d’arrêt,
  • définir une formule de récurrence.

```
fonction monter_escalier( h : entier )
début
si h>0
alors     monter_marche()
          monter_escalier(h-1)
fsi
fin
```

Recette de récursivité

S’assurer que le problème peut se décomposer en un ou plusieurs **sous-problèmes de même nature**

- Identifier le **cas de base (ou cas trivial) qui est le plus petit problème** qui ne se décompose pas en sous-problèmes

- Résoudre(P) =
  - si P est un cas de base, le résoudre directement
  - sinon
    - décomposer P en sous-problèmes P1, P2,...
    - résoudre récursivement P1, P2,...
    - combiner les résultats obtenus pour P1, P2, …, pour obtenir la solution pour avoir la solution au problème de départ.

Algorithme général

fonction F_récursive (paramètres)
début
si <condition d'arrêt> alors
        instructions résolvant directement les cas particuliers
sinon
        instructions
        appel(s) récursif(s) à F_récursive(paramètres modifiés)
        instructions
        (l'ordre peut changer selon les problèmes)
fsi
fin

Consignes

• Il faut noter que les paramètres passés à la fonction récursive doivent changer d'une étape à une autre, sinon les appels récursifs ne se termineront jamais. Ils sont généralement « plus simples » d'un appel au suivant et doivent modifier la condition d'arrêt au bout d'un temps fini.

• Il ne faut jamais mettre l'appel récursif en première instruction de l'algorithme récursif, sinon on aura un appel récursif infini.

Exemple : fonction monter_escalier(h : entier)

```
début
si h>0
alors     monter_marche()
          monter_escalier(h-1)
fsi
fin
```

Classes d’algorithmes

On peut distinguer 4 classes d’algorithmes récursifs en fonction des types de problèmes à résoudre et des solutions proposées :

- les problèmes définis par récurrence
- les recherches avec retour arrière
- les techniques de découpage du type « diviser pour régner »
- les problèmes liés aux structures d’informations récursives.

Nous étudierons des exemples dans chacune de ces classes.

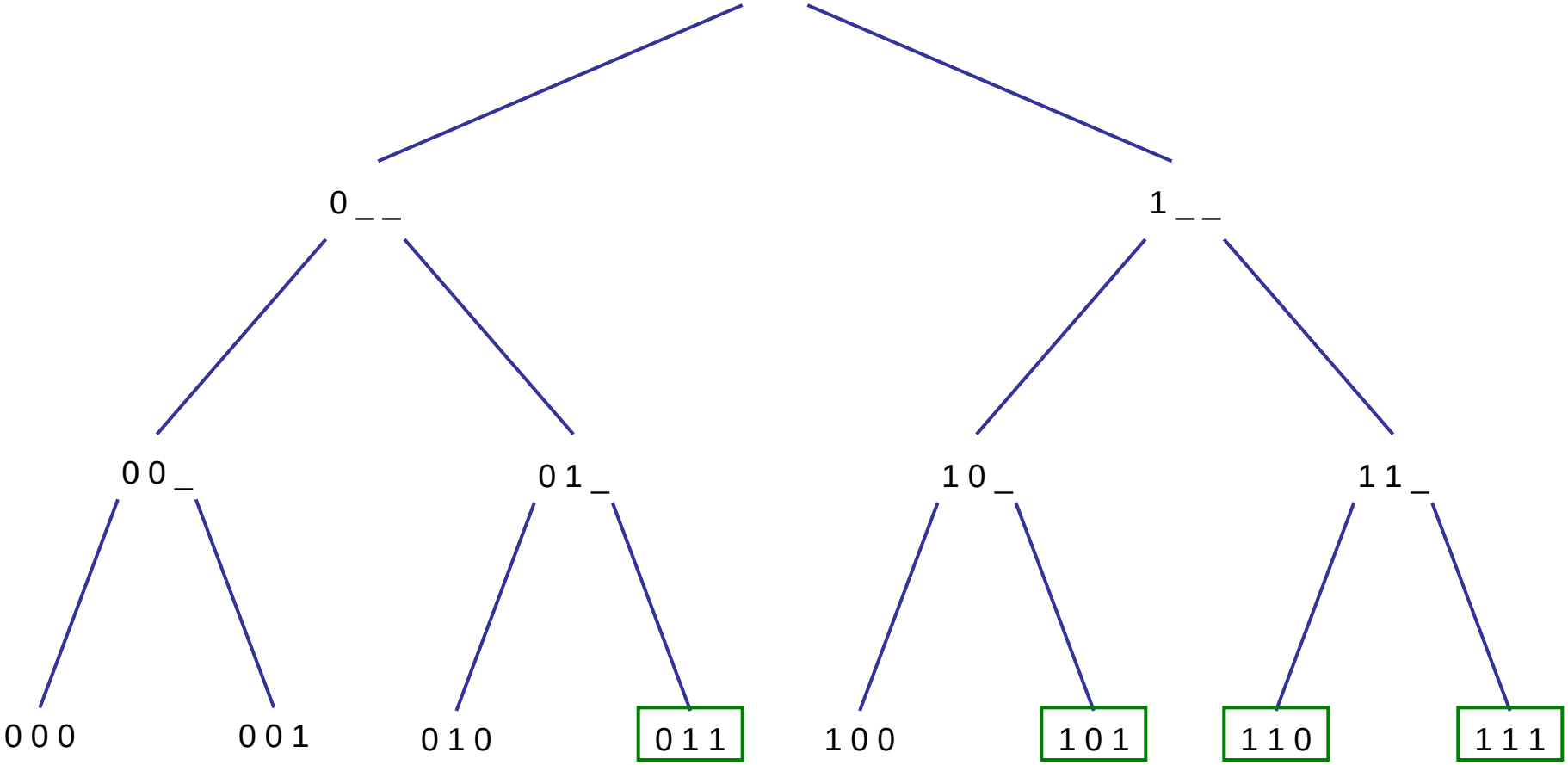
Algorithmique Avancée

Algorithmes de recherche avec retour arrière (Techniques de Backtracking)

Exemple illustratif

- Soient l'ensemble de tous les vecteurs binaires à 3 bits. Chercher tous les vecteurs dont la somme des « 1 » est supérieure ou égale à 2.
- Le seul moyen pour résoudre ce problème est de vérifier toutes les possibilités :
  (000, 001, 010, ...,111).
- Il y a huit (08) possibilités (espace de recherche). On peut les représenter sous forme d'un arbre.

Exemple illustratif (suite)



Recherche avec retour arrière : principe

- Faire des choix et prendre des décisions puis de remettre en question ces décisions afin d’éviter une impasse
- Revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage
- Idée : essayer chaque possibilité (combinaison) jusqu'à trouver la bonne

Recherche avec retour arrière : principe

- Pendant la recherche, si on essaie une alternative qui ne satisfait plus une contrainte, on effectue un retour arrière à un point où d'autres alternatives s'offraient à nous, et on essaie la possibilité suivante. Si on n'a plus de tels points la recherche échoue.
- Le point fort du « backtracking » est que beaucoup de ses réalisations évitent d'essayer un maximum de combinaisons partielles, diminuant ainsi le temps d'exécution.

Trouver une seule solution pour un problème donnée

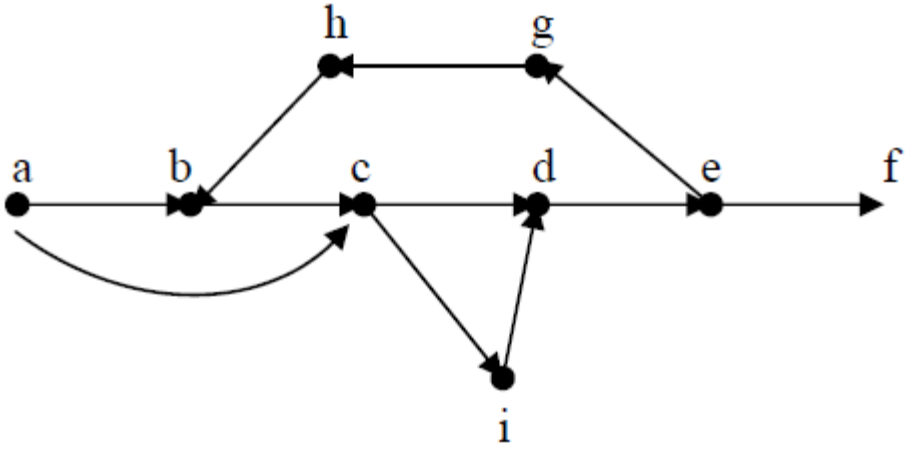
```
fonction rechercherSolution (…) : booléen
début
    si solution complète alors
        infructueux ← faux
    sinon
        infructueux ← vrai
        initialiser la sélection des candidats
        répéter
            choix d'un candidat
            si acceptable alors        (* Le candidat satisfait certaines conditions *)
                enregistrer            (* Le candidat est enregistré. Construction partielle de la solution *)
                infructueux ← rechercherSolution (…)
                si infructueux alors
                    effacer enregistrement
                fsi
            fsi
        jusqu'à ( non infructueux) ou (plus de candidats)
    fsi
    retourne infructueux
fin
```

Trouver toutes les solutions pour un problème donnée

```
fonction rechercherSolution (…)
début
    si solution complète alors
        imprimer la solution
    sinon
        initialiser la sélection des candidats
        pour chaque candidat
            si acceptable alors
                enregistrer
                rechercherSolution (…)
                effacer enregistrement
            fsi
        fpour
    fsi
fin
```

Recherche de chemins dans un graphe avec circuits

Exemple de graphe



Un graphe est un ensemble de *nœuds (ou sommets)* et de relations (ou arcs) entre ces nœuds. Le graphe donné en exemple a pour nœuds : a, b, c, d, e, f, g, h et i. Le nœud début du graphe est a et le nœud fin *f.* Un chemin commence en a, se termine en f et ne passe jamais 2 fois par le même nœud.

Les chemins du graphe donné en exemple sont les suivants :

a b c d e f, a b c i d e f, a c d e f, a c i d e f.

Recherche de chemins dans un graphe avec circuits

Rechercher tous les chemins dans un graphe avec circuits

Méthode de résolution

1. On construit les chemins un à un, nœud par nœud.
2. Si, à un instant donné, on se trouve au nœud *extrémité* qui est l'extrémité d'un début de chemin, on le complète en choisissant un des nœuds successeurs de *extrémité.*
3. Le nœud choisi ne doit pas déjà appartenir au chemin. Si c'est le cas et qu'il n'y a pas d'autre choix possible, on remet en question le choix de *extrémité (➔ retour arrière).*

rechercherChemin(début_chemin, extrémité, arrivée, graphe)

Algorithme informel

```
Algorithme informel
fonction rechercherChemin (début_chemin, extrémité, arrivée, graphe)
 début
    si extrémité = arrivée alors
            écrire (début_chemin)
    sinon
            Pour chaque nœud ∈ {successeur (extrémité)} faire
                    Si nœud ∉ début_chemin alors
                            début_chemin ← début_chemin + noeud
                            rechercherChemin (début_chemin + nœud, nœud, arrivée, graphe)
                            début_chemin ← début_chemin –  nœud
                    fsi
            fpour
    fsi
fin

Lexique :
        début_chemin : début d'un chemin
        extrémité : nœud extrémité de début_chemin
        graphe : le graphe dans lequel on cherche les chemins
        arrivée : nœud terminal du graphe (f dans l'exemple)
```

Pour trouver tous les chemins du graphe donné en exemple, on fait l'appel suivant : **rechercherChemin(a, a, f, graphe)**

Choix d'une représentation logique du graphe

Première possibilité :

Le graphe est représenté par une table dont les clés sont les couples de nœuds et les valeurs vrai ou faux :

Clé = {(i, j) / i,j sont des nœuds } et Valeur = {vrai, faux}

A un couple de nœud (i, j), graphe associe vrai si on peut aller du nœud i au nœud j directement (c'est-à-dire s'il y a un arc de i vers j) et faux sinon.

Le graphe exemple peut être schématisé de la manière suivante :

	a	b	c	d	e	f	g	h	i
a	faux	**vrai**	**vrai**	faux	faux	faux	faux	faux	faux
b	faux	faux	**vrai**	faux	faux	faux	faux	faux	faux
c	faux	faux	faux	**vrai**	faux	faux	faux	faux	**vrai**
d	faux	faux	faux	faux	**vrai**	faux	faux	faux	faux
e	faux	faux	faux	faux	faux	**vrai**	**vrai**	faux	faux
f	faux	faux	faux	faux	faux	faux	faux	faux	faux
g	faux	faux	faux	faux	faux	faux	faux	**vrai**	faux
h	faux	**vrai**	faux	faux	faux	faux	faux	faux	faux
i	faux	faux	faux	**vrai**	faux	faux	faux	faux	faux

Choix d'une représentation logique du graphe (suite)

Deuxième possibilité :

Le graphe est représenté par une table dont les clés sont les nœuds et les valeurs des listes de nœuds :

Clé = {Nœud} et Valeur = {liste (Nœud)}

A chaque nœud, graphe associe la liste de ses successeurs.

Ecrire l'algorithme formel en choisissant la première représentation possible.

Choix d'une représentation logique du graphe (suite)

Algorithme formel

```
fonction rechercherChemin (début_chemin : ListeNoeud, extrémité : Nœud, arrivée : Nœud,
graphe : Graphe, ens_nœud : EnsNoeud)
  début
      si extrémité = arrivée alors
            écrire (début_chemin)
      sinon
            Pour chaque nœud dans ens_noeud faire
                si accèstab (graphe, (extrémité, nœud)) alors
                      circuit ← faux
                      place ← tête (début_chemin)
                      tant que non finliste (début_chemin, place) et non circuit faire
                          si val (début_chemin, place) = nœud alors
                                circuit ← vrai
                          sinon
                                place ← suc (début_chemin, place)
                          fsi
                      ftant
                      si non circuit alors
                          adjqlis (début_chemin, nœud)
                          rechercherChemin (début_chemin, nœud, arrivée, graphe, ens_nœud )
                          supqlis(début_chemin)
                      fsi
                fsi
            fpour
      fsi
  fin
```

Lexique

ListeNœud = liste (Nœud)
début_chemin : ListeNœud, début du chemin courant
extrémité : Nœud , nœud extrémité de *début_chemin*
Graphe = table[(Nœud, Nœud) → booléen]
graphe : Graphe , le graphe dans lequel on cherche les chemins
arrivée : Nœud, nœud terminal du graphe
EnsNœud = Ens (Nœud)
ens_nœud : EnsNœud , ensemble des nœuds du graphe
nœud : Nœud, nœud courant du graphe
place : Place, dans *début_chemin*
circuit : booléen, à vrai si le nœud courant fait déjà partie de *début_chemin*

Complexité des algorithmes

Algorithmique Avancée

2ème Année DUT Informatique

Complexité des Algorithmes

- Plusieurs algorithmes peuvent exister pour résoudre un même problème.
- Quel algorithme choisir? Quel est le plus efficace?
- Pour déterminer l'efficacité d'un algorithme, il faut évaluer les ressources utilisées :
  - temps,
  - mémoire,
  - nombre de processeurs,
  - bande passante d'un réseau de communication,
  - etc.

Analyse de l'efficacité d'un algorithme en terme de temps

- La complexité temporelle est la plus importante
- La mesure de l'efficacité d'un algorithme doit être:
  - indépendante de la machine,
  - du langage de programmation,
  - du programmeur,
  - et tous les détails liés à l'implémentation.

Analyse de la complexité d'un algorithme

- Objectif :

  proposer des méthodes qui permettent d'estimer le coût d'un algorithme, de comparer deux algorithmes différents (sans avoir à les programmer effectivement)

- Coût d'un algorithme :

  déterminer une fonction associant un coût en unité de temps à un paramètre entier n qui résume la taille des données.

- Exemples :

  - recherche d'un élément dans un tableau : n est la taille du tableau,
  - tri d'une liste : n est la taille de la liste,
  - calcul d'un terme d'une suite de récurrence : n est l'indice du terme.

Différents types de complexité

- La complexité dans le pire des cas :

  → la plus utilisée

- La complexité en moyenne :

  → révèle le mieux le comportement « réel » de l'algorithme

  → difficile à calculer en général

- La complexité dans le meilleur des cas :

  → très peu utilisée

Exemple 1

- Algorithme:

```
pour j de 1 à n faire                    n fois
        x ← x + 3*j                      c
fpour
```

c: coût de l'instruction `x ← x + 3*j`

- Coût total de cet algorithme (ou sa complexité) :

$c \times n = c.n$

Exemple 2

- Algorithme:

```
pour j de 1 à n faire                    n fois
        x ← j * 2                        c1
fpour
pour i de 1 à n faire                    n fois
    pour j de 1 à n faire                n fois
        x ← x + i*x+j                    c2
    fpour
fpour
```

- Coût total de cet algorithme (ou sa complexité) :

$$n \times c_1 + n \times n \times c_2 = c_2 \,.\, n^2 + c_1.n$$

Ordre de grandeur

On s'intéresse plus à l'ordre de grandeur de la complexité qu'à son calcul exact.

- L'influence des coefficients (coûts des opérations élémentaires) est négligeable sur le coût final lorsque n est grand.

➔ *la complexité est indépendante de la machine*.

- Le monôme ayant la puissance la plus élevée est le plus important.

Exemple 1

- Algorithme:

```
pour j de 1 à n faire                    n fois
        x ← x + 3*j                      c
fpour
```

c: coût de l'instruction `x ← x + 3*j`

- Coût total de cet algorithme (ou sa complexité) :

$$c \times n = c.n$$

Cet algorithme a une complexité en O(n)

Exemple 2

- Algorithme:

```
pour j de 1 à n faire                    n fois
        x ← j * 2                        c1
fpour
pour i de 1 à n faire                    n fois
    pour j de 1 à n faire                n fois
        x ← x + i*x+j                    c2
    fpour
fpour
```

- Coût total de cet algorithme (ou sa complexité) :

$$n \times c_1 + n \times n \times c_2 = c_2 \,.\, n^2 + c_1.n$$

Cet algorithme a une complexité en O(n²)

Ordre de grandeur

Propriétés : (pour n suffisamment grand)

- Si $a < b$, alors un algorithme qui est en $O(n^b)$ a une complexité plus importante qu'un algorithme en $O(n^a)$

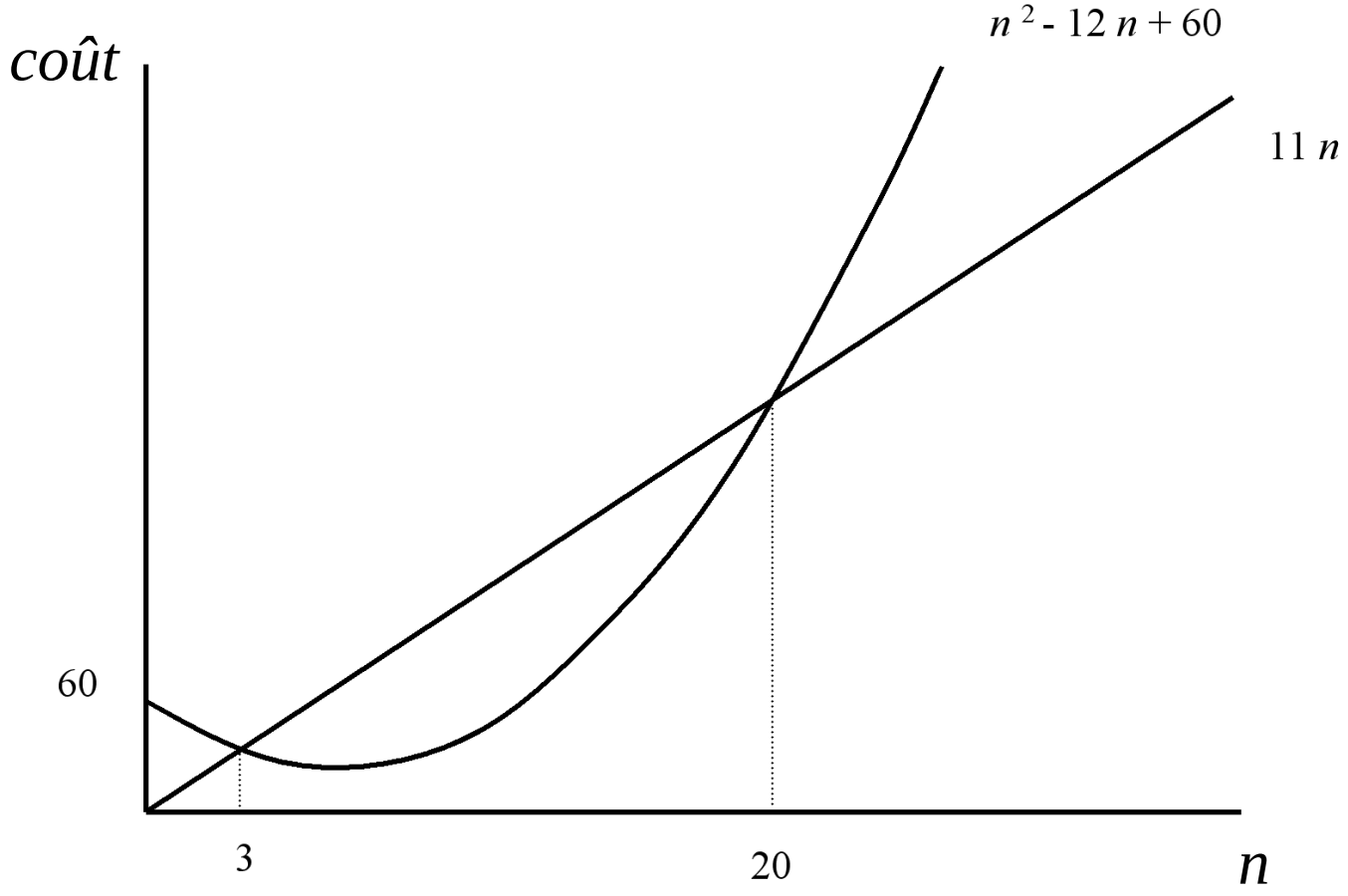
- Les algorithmes ayant une complexité en $\log n$ sont plus rapides que les algorithmes ayant une complexité en n^α ($\alpha > 0$).

- Les algorithmes ayant une complexité en n^k (pour tout k>0) sont plus rapides que les algorithmes ayant une complexité en c^n (pour $c > 0$)

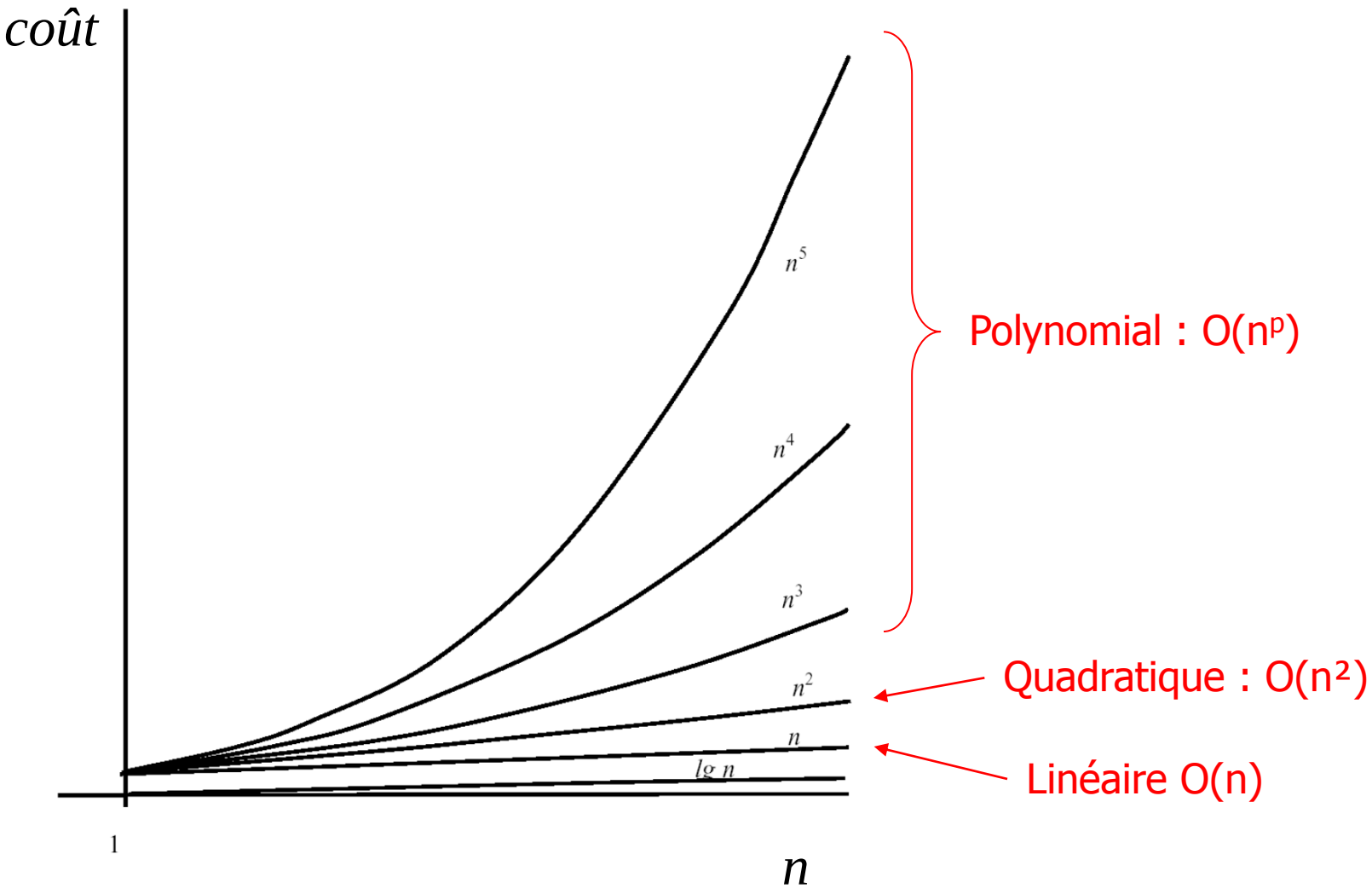
Ordre de grandeur

Complexité de A : $n^2 - 12n + 60 \Rightarrow O(n^2)$

Complexité de B : $11n \Rightarrow O(n)$



Exemples de croissance de fonctions



Les principales classes de complexité

Complexité logarithmique : coût en $O(log_2\ n)$

- recherche dichotomique dans un tableau trié $t[1..n]$.

Complexité linéaire : coût en $O(n)$

- factorielle n

Complexité quasi-linéaire : coût en $O(n\ log\ n)$

- tri par fusion, tri rapide

Complexité polynomiale : coût en $O(n^p)$, p > 1

- $p = 2$, tri par sélection

Complexité exponentielle : coût en $O(a^n)$ avec $a > 1$

- tours de Hanoï

Importance des ordres de grandeurs

algorithme	1	2	3	4	5
Coût en temps (micro sec.)	**33 n**	**46 n lg n**	**13 n^2**	**3.4 n^3**	**2^n**
Taille des données	**Temps de résolution**				
10	0.00033 sec	0.0015 sec	0.0013 sec	0.0034 sec	0.001 sec
100	0.003 sec	0.03 sec	0.13 sec	3.4 sec	4 .10^{16} années
1000	0.033 sec	0.45 sec	13 sec	0.94 heures	
10 000	0.33 sec	6.1 sec	22 min	39 jours	
100 000	3.3 sec	1.3 min	1.5 jours	108 années	

Importance des ordres de grandeurs

TRS-80

Si un algorithme a une complexité importante, même si on l'exécute sur machine très rapide, cela ne rendra pas l'algorithme plus rapide.

Importance des ordres de grandeurs

Taille des données	Cray-1	TRS-80
n	**3 n^3 nanosecondes**	**19 500 000 n nanosecondes**
10	3 microsecondes	0.2 secondes
100	3 millisecondes	2 secondes
1000	3 secondes	20 secondes
2 500	50 secondes	50 secondes
10 000	49 minutes	3.2 minutes
1 000 000	95 années	5.4 heures

2 ordinateurs des années 1977, Cray 1 plus rapide que TRS-80

Exemples de complexité d'algorithmes

Factorielle

```
fonction calculerFactorielle ( n : entier ) : entier
début
    si  n = 0 alors  factorielle ← 1
    sinon factorielle ← n * calculerFactorielle ( n-1 )
    fsi
    retourne factorielle
fin
```

$C(0) = 0$ (pas d'opérations arithmétiques)

1 est le coût de la multiplication

Exemples de complexité d'algorithmes

Factorielle

```
fonction calculerFactorielle ( n : entier ) : entier
début
    si  n = 0 alors  factorielle ← 1
    sinon factorielle ← n * calculerFactorielle ( n-1 )
    fsi
    retourne factorielle
fin
```

$C(0) = 0$ (pas d'opérations arithmétiques)

$C(n) = 1 + C(n\text{-}1)$ (1 est le coût de la multiplication)

$C(n) = 1 + C(n\text{-}1) = 2 + C(n\text{-}2) = 3 + C(n\text{-}3) = \ldots = n + C(0) = n$

Donc la complexité de factorielle est en ***O(n)*** ***(linéaire)***

Exemples de complexité d'algorithmes

Tours de Hanoï

```
fonction hanoï (n : entier, d : chaine, a : chaine, i : chaine)
début
  si n ≠ 0 alors
    hanoï (n -1, d, i, a )          C(n-1)
    écrire (« déplacer le disque », n , « de », d, « vers », a)   C = 1
    hanoï (n -1, i, a, d )          C(n-1)
  fsi
fin
```

Exemples de complexité d'algorithmes

Tours de Hanoï

```
fonction hanoï (n : entier, d : chaine, a : chaine, i : chaine)
début
   si n ≠ 0 alors
      hanoï (n -1, d, i, a )
      écrire (« déplacer le disque », n , « de », d, « vers », a)
      hanoï (n -1, i, a, d )
   fsi
fin
```

C(n-1) → hanoï (n -1, d, i, a)

C = 1 → écrire (« déplacer le disque », n , « de », d, « vers », a)

C(n-1) → hanoï (n -1, i, a, d)

$C(0) = 0$

$C(n) = 1 + 2 \times C(n-1)$

Par récurrence, $C(n) = 2^n - 1$

Donc la complexité de l'algorithme des tours de Hanoï est :

$$\boldsymbol{C(n) = O(2^n)} \qquad \textbf{(exponentiel)}$$

Exemples de complexité d'algorithmes

Recherche dichotomique

Algorithme

```
fonction rechercheDicho (tab : Tab, borne_inf : entier, borne_sup : entier, élément : Elément) : entier
début
    si borne_inf = borne_sup alors
            si tab[borne_inf] = élément   alors
                        indice ←borne_inf
            sinon
                        indice ← -1
            fsi
    sinon
        milieu ← (borne_inf + borne_sup) ÷ 2
        si élément ≤ tab[milieu] alors
            indice ←rechercheDicho (tab, borne_inf, milieu, élément)
        sinon
            indice ←rechercheDicho (tab, milieu + 1, borne_sup, élément)
        fsi
    fsi
    retourne indice
fin
```

Traitement non récursif

Coût = 1

Coût = 1

Appels récursifs

Exemples de complexité d'algorithmes

Recherche dichotomique

```
Algorithme
fonction rechercheDicho (tab : Tab, borne_inf : entier, borne_sup : entier, élément : Elément) : entier
début
    si borne_inf = borne_sup alors
            si tab[borne_inf] = élément  alors
                        indice ←borne_inf
            sinon
                        indice ← -1
            fsi
    sinon
        milieu ← (borne_inf + borne_sup) ÷ 2
        si élément ≤ tab[milieu] alors
            indice ←rechercheDicho (tab, borne_inf, milieu, élément)
        sinon
            indice ←rechercheDicho (tab, milieu + 1, borne_sup, élément)
        fsi
    fsi
    retourne indice
fin
```

Traitement non récursif

Coût = 1

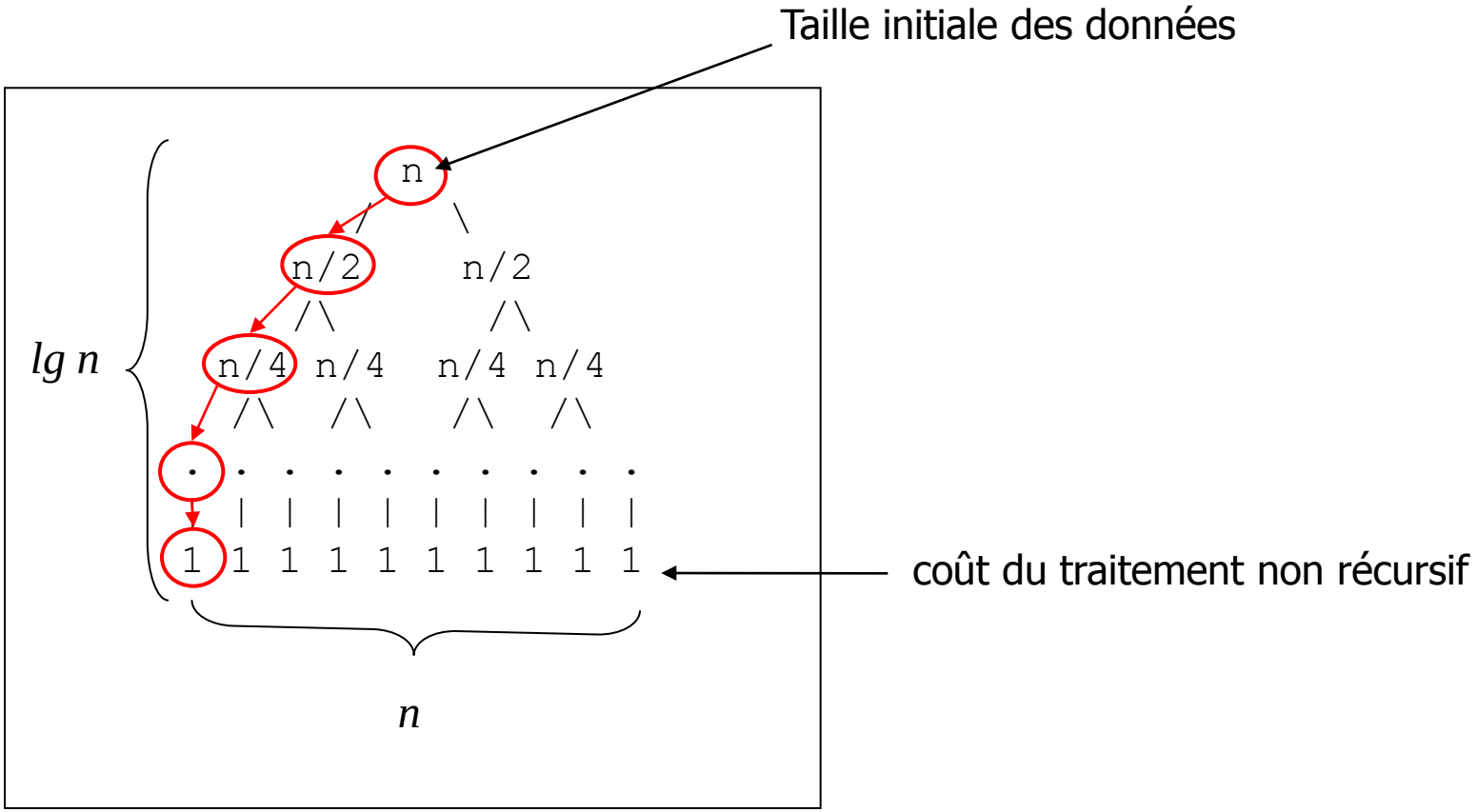
Coût = 1

Appels récursifs

$C(n) = C(n \quad 2) + 1 \quad \text{pour} \quad n > 1$

$C(1) = 1$

$\Rightarrow$ complexité de l'algorithme de rech. Dichotomique : $O(lg\ n)$


Taille initiale des données
n
n/2
n/2
n/4
n/4
n/4
n/4
lg n
1 1 1 1 1 1 1 1 1
coût du traitement non récursif
n