Cours3
Parcours et recherche

Année 2016-2017 J-F. Kamp - IUT de Vannes

PLAN

- Complexité :
  - Ordre de grandeur
  - Principales classes de complexité
  - Limites de la complexité

- Parcours et recherche
  - Exemples
  - Recherche séquentielle
    - Première solution

  - Recherche dichotomique
    - Hypothèses
    - Illustration
    - Complexité

Complexité : suite

Complexité théorique

```
int factorielle ( int n ) {

    int ret, i:                     // 2 op

    ret = 1;                        // 1 op
    i = 1;                          // 1 op

    while ( i <= n ) {              // 1 op

        ret = ret * i;              // 2 op
        i++;                        // 2 op
    }

    return ret;
}
```

- Avant la boucle : 2 + 1 + 1 = 4 op
- Boucle effectuée (n – 1 + 1) = n fois
- Corps de la boucle : 2 + 2 = 4 op
- Comparaison : 1 op
- Comparaison effectuée (n + 1) fois

Ordre de grandeur

Nbre opérations élémentaires

$f(n) = 5 + 5n$

L'expression **exacte** n'a pas beaucoup d'intérêt car :

- La constante « 5 » n'est pas significative elle concerne des détails d'implémentation.
- Le coût en temps sera d'autant plus important que **n** est grand et la constante n'a plus aucun poids dans le calcul.

On fera l'approximation suivante : pour **n** grand $\mathbf{f(n) \approx 5n}$, donc la complexité de l'algorithme « factorielle » est linéaire en **n**, ce que l'on notera $\mathbf{\Theta(n)}$.

Classes de complexité

Θ	Noms	Exemples
Θ(1)	constante	tab[i]
Θ(logn)	logarithmique	rech. dichotom.
Θ(n)	linéaire	factorielle
Θ(nlogn)	nlogn	tris rapide
Θ(n^2)	quadratique	tris simple
Θ(n^3)	cubique	3 bcles imb.
Θ(2^n)	exponentielle	IA (cavalier)

Limites de la complexité

Pour **n** petit, les constantes cachées peuvent faire la différence.

Exemple :

Soit algo1 : $f(n) = 20 + 100\, n \log_2(n)$

Soit algo2 : $f(n) = 40 + 2\, n^2$

algo1 est en $\Theta(n\log n)$

algo2 est en $\Theta(n^2)$

Pour n = 8 (donc petit) on a :

algo1 : $f(8) = 100 \times 8 \times 3 + 20 = 2420$ op

algo2 : $f(8) = 40 + 2 \times (8)^2 = 168$ op

Qui est le meilleur ?

Parcours et recherche

Exemples

La recherche est un problème classique de tous les « jours » :

- rechercher un correspondant téléphonique
- rechercher un fichier dans un répertoire
- rechercher un mot-clé dans un document
- taper une requête sur Google

Exemples

Cette recherche se fait dans ce que l’on appelle une « collection » :

- collection de correspondants
- collection de fichiers
- collection de mots
- collection d’URLs

Exemples

La collection la + simple en ce qui nous concerne, c’est le tableau d’entiers de taille N.

29	50	-56	205	129	1000
0	1	2	3	4	5

Rechercher l’entier « 129 », c’est parcourir tout le tableau et s’arrêter lorsque l’on tombe sur la case contenant « 129 ».

Recherche séquentielle

Première solution

Hypothèses :

- Un tableau de taille N, rempli partiellement avec **nb** entiers
- Le tableau contient AU PLUS un entier égale à la valeur à chercher

Boucle sur le tableau et comparaison des éléments un par un.

Conditions d’arrêt :

- Soit l’entier est trouvé (**trouve == true**)
- Soit on atteint le dernier élément du tableau (**i == nb**)

Première solution

Conditions d’arrêt (deux) :

- Soit l’entier est trouvé (**trouve == true**)
- Soit on atteint le dernier élément du tableau (**i == nb**)

Condition de continuation :

- Loi de Morgan

  non (A ou B) = non(A) et non(B)
- **(trouve == false) && (i != nb) ou bien (i < nb)**

En Java

En Java, méthode « rechercheSeq ».

```
/* *
   *  Cette méthode recherche une valeur dans un
   *  tableau d'entiers. Elle renvoie le numéro de la
   *  case contenant cet entier ou -1 si il n'existe pas.
   *  @param leTab le tableau des valeurs
   *  @param aRech valeur à rechercher
   *  @param nb le nbre de valeurs
   *  @return le numéro de la case (ou -1)
   */
int rechercheSeq ( int [ ] leTab, int nb, int aRech ) {
   // variables locales
   int i, ret ;
   boolean trouve ;
   // initialisations
   // boucle
   // terminaison
   return ret ;
}
```

En Java

En Java, méthode « rechercheSeqV1 ».

```
int rechercheSeqV1 ( int [ ] leTab, int nb, int aRech ) {

    // variables locales
    int i, ret;
    boolean trouve ;

    // initialisations
    i = 0;
    trouve = false;
    ret = -1;

    // boucle
    while ( ( trouve == false ) && ( i < nb ) ) {

        if ( aRech == leTab[i] ) {
            trouve = true;
            ret = i;
        }

        i++;
    }


    return ret ;
}
```

En Java

En Java, méthode « rechercheSeqV2 ».
Simplifions : pas besoin de « ret », …

```
int rechercheSeqV2 ( int [ ] leTab, int nb, int aRech ) {

    // variables locales
    int i ;
    boolean trouve ;

    // initialisations
    i = 0;
    trouve = false;

    // boucle
    while ( !trouve && ( i < nb ) ) {

        if ( aRech == leTab[i] ) {
            trouve = true;
        }

        else {
            i++;
        }
    }

    // terminaison
    if ( !trouve ) i = -1;

    return i ;
}
```

En Java

En Java, méthode « rechercheSeqV3 ».
Simplifions : pas besoin de « ret », remplacer la condition « trouver » par break.

```
int rechercheSeqV3 ( int [ ] leTab, int nb, int aRech ) {

    // variables locales
    int i ;

    // initialisations
    i = 0;

    // boucle
    while ( i < nb ) {

        if ( aRech == leTab[i] ) {
            break;
        }

        i++;
    }

    // terminaison
    if ( i == nb ) i = -1;

    return i ;
}
```

Analyse en complexité

Que vaut f (nb) ?

- Dans le meilleur des cas ?

  La valeur se trouve en position zéro.

- Dans le pire des cas ?

  La valeur ne s’y trouve pas.

- Dans le cas moyen ?

  La valeur se trouve à la moitié du tableau (nb/2).

Analyse en complexité dans le pire des cas

Hypothèses habituelles :

- Une déclaration : 1 opération élémentaire
- Une affectation : 1 opération élém.
- Une addition ou multiplication : 1 opération élém.
- Une incrémentation (i++) ou décrémentation (i--) : 2 opérations élém.
- Un test de comparaison (<=, <, >=, >, !=, ==) : 1 opération élém.
- Accès à 1 élément de tableau tab[i] : 0 opération élém.
- break : 0 opération élém.

Complexité de la version3

```
int rechercheSeqV3 ( int [ ] leTab, int nb, int aRech ) {

    // variables locales
    int i ;                                  // 1 op
    // initialisations
    i = 0;                                   // 1 op
    while ( i < nb ) {                       // 1 op

        if ( aRech == leTab[i] ) {           // 1 op
            break;                           // 0 op
        }

        i++;                                 // 2 op
    }
    if ( aRech != leTab[i] ) {               // 1 op
        i = -1;                              // 1 op
    }

    return i ;
}
```

- Avant la boucle : 1 op
- Init. de la boucle : 1 op
- Boucle effectuée (nb – 1 + 1) = nb fois
- Cond. continuation : 1 op
- Corps de la boucle : 1 op
- Incrémentation : 2 op
- Terminaison : 2 op (pire des cas)

Complexité de la version3 dans le pire des cas

Nbre opérations élémentaires = f (nb)

= 1 + 1 + (nb + 1) x 1 + (nb x 3) + 2

= 5 + 4nb

Soit nb = n (taille du problème)

L'algorithme est linéaire en **n**. On dira que l'algorithme est en **Θ(n)**.

Commentaires :

- Rien d'étonnant : 1 boucle => classe de complexité **n**.
- Essai d'amélioration de l'algorithme pour diminuer la constante **4** (devant n).

Recherche dichotomique

Hypothèse de départ

Le gain sur l'efficacité de la recherche repose sur une structure de départ particulière.

La collection dans laquelle on effectue la recherche est TRIEE.

Si elle est triée alors la recherche se fera beaucoup plus rapidement et sera < **Θ(n)**.

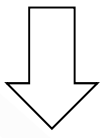
Exemple

-56	29	50	129	205	1000
0	1	2	3	4	5

Le tableau est trié par ordre croissant.

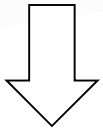
Supposons la recherche de « 129 ».

Au départ on « coupe » le tableau en deux parties « égales » : (0 + 5)/2 = 2 (partie entière).



-56	29	50	129	205	1000
0	1	2	3	4	5

Exemple



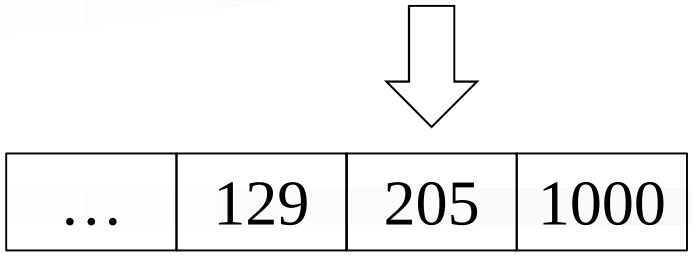
-56	29	50	129	205	1000
0	1	2	3	4	5

Comme tab[2] = 50 < 129, on est certain que « 129 » EST dans la partie du tableau à droite de l'indice 2 entre les indices 3 et 5.

La recherche se poursuit alors uniquement dans ce « nouveau » tableau.

…	129	205	1000
	3	4	5

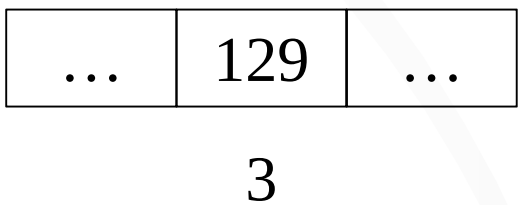
Exemple



On recoupe le tableau en deux : (3 + 5)/2 = 4.

Comme tab[4] = 205 > 129, on est certain que « 129 » EST dans la partie du tableau à gauche de l'indice 4 entre les indices 3 et 3.

La recherche se poursuit alors uniquement dans ce « nouveau » tableau à 1 seule case => on a trouvé.



Exemple

-56	29	50	129	205	1000
0	1	2	3	4	5

Nombre de boucles pour trouver « 129 » ?

Soit « indM » l'indice milieu du sous-tableau :

- Etape 1 : indM = 2
- Etape 2 : indM = 4
- Etape 3 : indM = 3

Méthode séquentielle : 4 étapes

Est-ce vraiment intéressant ? Oui pour n ↗

L’algorithme

En Java, méthode « rechercheDicho».

```
int rechercheDicho ( int [ ] leTab, int nb, int aRech ) {
   // variables locales
   // indD = indice de début du sous-tableau
   // indF = indice de fin du sous-tableau
   // indM = indice milieu entre indD et indF
   int indD, indF, indM, ret;

   // initialisations
   indD = 0;
   indF = nb – 1;

   // boucle
   // terminaison
   return ret;
}
```

L’algorithme

```
int rechercheDicho ( int [ ] leTab, int nb, int aRech ) {
    // variables locales
    …
    // initialisations
    indD = 0;
    indF = nb – 1;
    // boucle
    while ( indD != indF ) {
        indM = ( indD + indF ) / 2;    // division entière !
        if ( aRech > leTab[indM] ) {
            indD = indM + 1;
        }
        else indF = indM;
    }
    // terminaison
    return ret;
}
```

L’algorithme

```
int rechercheDicho ( int [ ] leTab, int nb, int aRech ) {
    // variables locales
    …
    // initialisations
    …
    // boucle
    while ( indD != indF ) {
        indM = ( indD + indF ) / 2;    // division entière !
        if ( aRech > leTab[indM] ) {
            indD = indM + 1;
        }
        else indF = indM;
    }
    // terminaison : indD == indF forcément
    // MAIS leTab [indD] par forcément = aRech !!
    if ( aRech == leTab [indD] ) ret = indD;
    else ret = -1;
    return ret;
}
```

Analyse en complexité

Que vaut f (nb) ?

- Dans le meilleur des cas ?
- Dans le pire des cas ?
- Dans le cas moyen ?

=> AUCUNE différence pour la recherche dichotomique

Analyse en complexité

```
int rechercheDicho ( int [ ] leTab, int nb, int aRech ) {
    int indD, indF, indM, ret;                          // 4 op

    indD = 0;                                           // 1 op
    indF = nb – 1;                                      // 2 op

    while ( indD != indF ) {                            // 1 op
        indM = ( indD + indF ) / 2;                     // 3 op
        if ( aRech > leTab[indM] ) {                    // 1 op
            indD = indM + 1;                            // 2 op
        }
        else indF = indM;                               // 1 op
    }
    if ( aRech == leTab [indD] ) ret = indD;            // 2 op
    else ret = -1;                                      // 1 op

    return ret;
}
```

Analyse en complexité

Combien de fois la boucle while (indD != indF) est exécutée ?

Supposons aRech se trouve en case zéro.

=> indD reste égale à 0 et à chaque itération indF = indM

Au départ indF – indD = n – 0 = n

Itération **1** : indM = n / 2 = n / $2^{\mathbf{1}}$

indM – indD = n / $2^{\mathbf{1}}$

Itération **2** : indM = n / 4 = n / $2^{\mathbf{2}}$

indM – indD = n / $2^{\mathbf{2}}$

Itération **3** : indM = n / 8 = n / $2^{\mathbf{3}}$

indM – indD = n / $2^{\mathbf{3}}$

… …

Itération **k** : indM = n / n = n / $2^{\mathbf{k}}$ = 1

indM – indD = 1 = n / $2^{\mathbf{k}}$

Analyse en complexité

Combien de fois la boucle while (indD != indF) est exécutée ?

k est le nombre de fois que la boucle est exécutée.

Itération **k** : $indM = n / n = n / 2^{\mathbf{k}} = 1$

$indM - indD = 1 = n / 2^{\mathbf{k}}$

Que vaut **k** = ?

Reste à résoudre : $1 = n / 2^{\mathbf{k}}$

$2^{\mathbf{k}} = n$

$$\Rightarrow k = \log_2 n$$

Complexité de la recherche dichotomique

Nbre opérations élémentaires = f (nb)

= 4 + 1 + 2 + (k x 5) + (k + 1) x 1 + 2

= 10 + 6k = 10 + 6 $\log_2$(nb)

Soit nb = n (taille du problème)

L'algorithme est logarithmique en **n**. On dira que l'algorithme est en **Θ(log n)**.

Commentaires :

Intéressant surtout pour n ↗

Soit n = 2^{20} = 1.048.576

Recherche séquentielle :

f(n) = 5 + 4n ≈ **4.000.000**

Recherche dichotomique :

f(n) ≈ 6 $\log_2$(n) = **120** (!!)