ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Analyse de la complexité algorithmique (1)

L'analyse de la complexité telle que nous l'avons vue jusqu'à présent nous a essentiellement servi à déterminer si un problème est ou non facile (i.e. soluble par un algorithme de complexité polynomiale).

Cependant, lorsqu'on s'intéresse à des problèmes polynomiaux (et c'est majoritairement le cas en informatique classique), il peut souvent y avoir ***plusieurs algorithmes polynomiaux*** pour résoudre ***un problème donné***.

Dans ce cas, il faut faire une analyse plus fine de leur complexité algorithmique, de façon à pouvoir choisir ***le plus performant***.

Analyse de la complexité algorithmique (2)

Comme la résolution algorithmique elle-même,
l'analyse de complexité des algorithmes

est une **tâche difficile**

pour laquelle il n'existe malheureusement
pas de recette générale.

Pour cette raison, nous allons tout d'abord étudier deux exemples
concrets qui nous servirons à illustrer quelques «***trucs et astuces***»

Analyse de la complexité: exemple (1)

Cas de la recherche d'un élément dans une liste triée

Soit l'algorithme naïf suivant :

```
appartient1(x,E) {
   for (i=0; i<taille(E); i++) { if (x==E[i]) return (true); }
   return (false);
}
```

et l'algorithme de recherche par dichotomie déjà vu (légèrement reformulé ici):

```
appartient2(x,E) { dichotomie(x,E,0,taille(E)); }

dichotomie(x,E,i,j) {
   if (i>j) return(false);
   else {
      k = (i+j)/2; u = E[k];
      if (x==u) return(true);
      else {
         if (x<u) return(dichotomie(x,E,i,k-1);
         else return(dichotomie(x,E,k+1,j);
}  }  }
```

Analyse de la complexité: exemple (2)

Pour calculer l'ordre de grandeur
de la complexité algorithmique d'algorithmes spécifiques,
plusieurs simplifications peuvent être faites :

1. **Mesure de la taille des entrées**:
   pour une entrée constituée de *m* données d'entrée, la taille à prendre dans l'approche théorique est la taille de la séquence binaire permettant de coder ces m données d'entrée. Cette taille étant dans la plupart des cas de la forme $m \cdot A+B$ où A et B sont des constantes, on pourra ***utiliser m comme mesure de la taille des entrées***, et donc exprimer la complexité par rapport à *m* (en effet $O((m \cdot A+B)^k) = O(m^k)$).

2. **Mesure de la complexité temporelle** :
   Dans l'approche théorique, la complexité temporelle est mesurée par le nombre de déplacements effectués par le tête de lecture/écriture lors d'exécution d'une machine de Turing représentant l'algorithme. Dans la pratique, cette machine de Turing est rarement disponible... et l'on mesure la complexité par le ***nombre d'instructions élémentaires nécessaires à l'exécution de l'algorithme***.

Analyse de la complexité: exemple (3)

Par ***instruction élémentaire***, nous entendrons ici
toute instruction qui peut être réalisée par une machine de Turing
en un nombre constant de déplacements de la tête de lecture/écriture.

Exemples d'instructions élémentaires:

- écrire un caractère à l'écran;
- lire un caractère dans un fichier;
- affecter une valeur atomique (un caractère, un entier, un réel, ...) à une variable (attention: l'affectation d'une valeur composée peut ne pas correspondre à un nombre constant de déplacements en cas dépendance par rapport au nombre d'éléments constituant la valeur composée);
- réaliser une opération arithmétique sur des valeurs atomiques;
- ...

Analyse de la complexité: exemple (4)

Analyse de la complexité de `appartient1(x,E)` :

Taille des entrées :

les entrées sont constituées d'un élément x et d'une liste de m=taille(E) éléments du même type que x. Que x soit atomique ou composé, la taille d'un codage binaire de la séquence d'entrée pourra être de la forme : $m \cdot A+B$ et l'on pourra prendre m comme mesure de référence.

Instructions élémentaires utilisées :

(1) l'affectation d'une valeur à une variable entière;
(2) le calcul de la taille d'une liste d'entiers;
(3) la comparaison de deux valeurs entières (avec < et ==);
(4) l'incrément d'une valeur entière;
(5) l'accès au i-ème élément d'une liste;
(6) le renvoi d'une valeur booléenne.

Analyse de la complexité: exemple (5)

Remarques a propos des instructions élémentaires:

- Pour des entiers codés sur un nombre fixe de bits, les instructions (1), (2), (3) et (6) peuvent être réalisées en un nombre constant de déplacements d’une machine de Turing.

  -> Ces instructions pourront donc être associées à un coût unité lors du calcul de complexité.

- Par contre, pour les instructions (2) et (5), la situation est plus compliquée, car dépendante de la représentation qui est utilisée pour les listes:

  - En effet, selon que cette représentation contient ou non de façon explicite l’indication de la taille, le calcul de cette dernière pourra être réalisé:
    -> soit en un nombre constant d’étapes (mémorisation explicite de la taille)
    -> soit en un nombre d’étapes dépendant du nombre d’éléments de la liste (parcours + sommation)

  - De même, selon que la liste est représentée ou non par une structure de données permettant un accès direct à ses éléments (tableau, vecteur), l’accès au $i^{ème}$ élément pourra se faire:
    -> soit en un nombre constant d’étapes (accès direct)
    -> soit à nouveau en un nombre d’étapes dépendant du nombre d’éléments de la liste (parcours).

Analyse de la complexité: exemple (6)

Dans le cas où la liste E est représentée par un structure de données permettant le calcul de la taille de E et l'accès à son i-ème élément en un temps constant, l'analyse de la complexité de `appartient1(x,E)` est la suivante:

déroulement de l'algorithme		instructions élémentaires
1	affectation de la valeur 0 à la variable i	1 instruction
2	calcul de la taille de E et vérification de la condition (i<taille(E))	2 instructions
3	accès au i-ème élément de E et comparaison de cet élément avec x	2 instructions
4	décrément (de 1) de i et retour en 2	1 instruction

Par construction de l'algorithme, les étapes 2, 3 et 4 seront faites autant de fois qu'il y a d'éléments dans E, donc m=taille(E) fois.

Si C1(m) est le nombre d'instructions élémentaires nécessaires pour réaliser l'algorithme `appartient1(x,E)`, on a donc:

$$C1(m) = O(2 + 5m) = \mathbf{O(m)}$$

Analyse de la complexité: exemple (7)

Dans le cas où la liste E est représentée par un structure de données ne permettant pas le calcul de la taille de E et l'accès à son i-ème élément en temps constant, l'analyse de la complexité de `appartient1(x,E)` est la suivante:

Supposons par exemple que E est représentée par une ***liste chaînée***. Si l'on ne fait pas d'efforts supplémentaires, le calcul de taille(E) et l'accès à E(i) seront alors tous les deux en $a+b\cdot m$ instructions (donc en $O(m)$).

Dans ce cas, les nombres d'instructions élémentaires pour les étapes seront les suivants :

pour l'étape 1:	1 instruction
pour l'étape 2 :	$a+b\cdot m$ instructions
pour l'étape 3 :	$a+b\cdot m$ instructions
pour l'étape 4 :	1 instruction

et on aura donc :

$$C1(m) = O(1 + m\cdot(2\cdot(a+ bm) + 1)) = O(2bm^2 + (2a+1)m + 1)$$

$$= \mathbf{O(m^2)}$$

Analyse de la complexité: exemple (8)

L'exemple du calcul de la complexité de `appartient1()` illustre le fait qu'**il faut être très *prudent pour la définition* de ce qui sera considéré comme *instruction élémentaire*** lors de l'analyse de complexité.

D'autre part, il faut aussi remarquer qu'**une petite modification** de l'algorithme peut avoir **un impact important sur sa complexité**.

Par exemple, si l'on reprend l'analyse de `appartient1(x,E)` en supposant l'accès aux éléments de E en temps constant mais la détermination du nombre d'éléments en $a+b \cdot m$ (cas de la représentation de E par un tableau statique).

Dans ce cas, les nombres d'instructions élémentaires pour les étapes seront les suivants :

pour l'étape 1:	1 instruction
pour l'étape 2 :	$a+b \cdot m$ instructions
pour l'étape 3 :	2 instructions
pour l'étape 4 :	1 instruction

et on aura à nouveau:

$$C1(m) = O(1 + m(a+ bm + 2) + 1) = \mathbf{O(m^2)}.$$

Analyse de la complexité: exemple (9)

Mais si l'on modifie très légèrement l'algorithme de la façon suivante :

```
appartient11(x,E) {
   for (i=0, j=taille(E); i<j; i++) { if (x==E[i]) return (true); }
   return (false);
}
```

Les nombres d'instructions élémentaires pour les étapes seront alors :

pour l'étape 1: a+b·m instructions
pour l'étape 2 : 1 instructions
pour l'étape 3 : 2 instructions
pour l'étape 4 : 1 instruction

et on aura donc:

$$C11(m) = O(a+bm + m(1 + 2 + 1)) = O(a + m(b+4)) = \mathbf{O(m)}.$$

Analyse de la complexité: exemple (10)

Analyse de la complexité de `appartient2(x,E)` :

Taille des entrées :
les entrées d'`appartient2(x,E)` sont les mêmes que celles d'`appartient1(x,E)`. On pourra donc à nouveau prendre *m* comme mesure de référence.

Instructions élémentaires utilisées :
les instructions élémentaires utilisées pour `appartient2()` sont du même type que celles utilisées pour `appartient1()`. La seule nouveauté est l'apparition d'**appels à des fonctions**. Pour les prendre en compte, il faut se souvenir du mécanisme mis en œuvre par le compilateur pour les réaliser:
- calcul des arguments;
- association des valeurs obtenues à des variables locales à la fonction;
- exécution du corps de la fonction.

Déroulement de l'algorithme:
du fait de la présence de ***structures de contrôle***, le déroulement de `appartient2()` est plus complexe que celui de `appartient1()`. C'est à ce niveau que l'on va utiliser le fait que l'on cherche à calculer une **complexité *pire cas*** : **en cas de choix**, ***on optera systématiquement pour la possibilités la plus défavorable***.

Analyse de la complexité: exemple (11)

Pour l'algorithme `appartient2(x,E)`,

```
1.    appartient2(x,E) { dichotomie(x,E,0,taille(E));}
2.    dichotomie(x,E,i,j) {
3.       if (i>j) return(false);
4.       else {
5.          k = (i+j)/2; u = E[k];
6.          if (x==u) return(true);
            else {
7.             if (x<u) return(dichotomie(x,E,i,k-1);
8.             else return(dichotomie(x,E,k+1,j);
      }  }  }
```

il existe des choix aux lignes 3, 6 et 7.

-> Les choix en 3 et 6 sont faciles à traiter, car le cas le plus défavorable est toujours celui correspondant à la valeur `false` de la condition. De ce fait, ***tant que la valeur `false` sera possible, le branchement correspondant sera emprunté***.

Une conséquence de cela est que la condition (x==u) ne devra jamais pouvoir être vérifiée... ce qui indique que la situation pire cas pour tester l'appartenance d'un élément x à une liste E est, du moins avec cette version de l'algorithme, ***lorsque x n'appartient pas à E***.

Analyse de la complexité: exemple (12)

Il nous reste donc à analyser le choix en 7:

-> Nous sommes ici dans un cas relativement facile; x n'appartenant pas à E, les deux possibilités associées à cette alternative – «`return(dichotomie(x,E,i,k-1)`» et «`return(dichotomie(x,E,k+1,j))`» – ne diffèrent donc que par les valeurs de i et de j (qui définissent la partie de E à considérer);

le cas le plus défavorable sera donc celui qui correspond à la sous-liste de E la plus grande.

Or, pour i et j entiers positifs, [i,((i+j)/2)-1] est toujours plus petit ou égal à [((i+j)/2)+1,j] et la possibilité la plus défavorable sera donc toujours la seconde (correspondant donc à x ≥ u)

En conclusion, le cas le plus défavorable pour `appartient2(x,E)` **est lorsque x n'appartient pas à E et est supérieur à tous ses éléments**.

Analyse de la complexité: exemple (13)

Le déroulement de l'algorithme est alors le suivant :

1. calcul de taille(E) et copie de x, E, 0 et taille(E);
2. comparaison (==) de deux valeurs entières (i et j);
3. calcul de k=(i+j)/2 ; accès au k-ème élément de E et affectation de sa valeur à u;
4. comparaison (==) de x avec u;
5. comparaison (<) de x, calcul de k+1, copie de x, E, k+1 et j et retour en 2.

Comme [k+1,j] est toujours au plus aussi grand que la moitié de [i,j],
la séquence d'étapes 2-5 sera donc réalisée **autant de fois que m=taille(E) peut être divisé par 2**, soit la partier entière de $\log_2(m)$ (qui sera notée $[\log_2(m)]$).

Analyse de la complexité: exemple (13)

Si l'on suppose que la liste E est représentée par un structure de donnée permettant le calcul de la taille de E et l'accès à son i-ème élément en un temps constant, les nombres d'instructions élémentaires pour les étapes seront alors :

pour l'étape 1:	5 instructions
pour l'étape 2 :	1 instructions
pour l'étape 3 :	3 instructions
pour l'étape 4 :	1 instruction
pour l'étape 5 :	6 instructions

et on aura donc :

$$C_2(m) = O(5 + [\log_2(m)](1+3+1+6)) = O(11*[\log_2(m)] + 5) = \mathbf{O(\log_2(m))}.$$

Les algorithmes `appartient1()` et `appartient2()` sont donc tous deux efficaces...

mais le second est tout de même substantiellement plus efficace que le premier.

Cas des boucles imbriquées (1)

Considérons l'algorithme de tri suivant :

```
   tri(E) {
      for (i=0; i<(taille(E)-1); i++) {
         MinI = i;
         for (ii=i+1;ii<taille(E); ii++) {
            if (E[ii]<E[i]) MinI = ii;
         }
         MinVal=E[MinI];
         E[MinI]=E[i];
         E[i]=MinVal;
}  }
```

Taille des entrées :

elle sera mesurée par m = taille(E).

Instructions élémentaires utilisées :

taille(E), E[i], affectation, comparaison, opérations arithmétiques.

Déroulement *pire cas* de l'algorithme:

chaque itération «for» est parcourue complètement et l'on suppose que la condition (E[ii]<E[i]) est toujours vérifiée.

Cas des boucles imbriquées (2)

D'où:

- pour la boucle intérieure:
  7 instructions élémentaires par itération,
  et la boucle contient $1+(m-1)-(i+1) = m-1-i$ itérations,
  soit un total de $n(i,m) = 7\cdot(m-1-i)$ instructions élémentaires.
- pour la boucle extérieure:
  8 instructions + les $n(i,m)$ instructions de la boucle intérieure,
  et la boucle contient toutes les itérations de $i=0$ à $i=m-2$ (soit $m-1$ itérations).

Le nombre total d'instructions élémentaires est donc:

$$8(m-1) + n(0,m)+n(1,m)+\ldots+n(m-2,m)$$

mais $n(0,m)+n(1,m)+\ldots+n(m-2,m) = 7((m-1)+(m-2)+\ldots+2+1) = 7(m(m-1)/2) = 7m(m-1)/2$

La complexité (pire cas) de `tri()` est donc :

$$C(m) = O(\ 8(m-1) + 7m(m-1)/2) = \mathbf{O(m^2)}.$$

Autre exemple: PGCD - Euclide (1)

Nous allons illustrer les notions présentées jusqu'ici sur la base d'un second exemple: ***l'algorithme d'Euclide*** pour le **calcul du plus grand commun diviseur de deux entiers**.

Une formulation possible (ici la formulation récursive) de l'algorithme est:

$\forall$ `i,j` $\in$ N, `i` $\geq$ `j` $\geq$ `2`

```
PGCD(i,j) {
   if (j==0) return (i);
   else return(PGCD(j, i % j));
}
```

Notations:

`x ÷ y` -> la division entière de `x` par `y`
`x % y` -> reste de la division entière de `x` par `y`
`x / y` -> la division réelle de `x` par `y`

Le déroulement pire cas de `PGCD` est:

1. tester l'égalité entre `j` et `0`;
2. calculer `(i % j)` – et copier cette valeur ainsi que `j` dans respect. `j` et `i`;
3. revenir en (1).

On voit ainsi que chaque boucle (1,2,3) correspond à un nombre fixe (ici 4) d'instructions élémentaires. De ce fait, si I est le nombre de fois que la boucle (1,2,3) est effectuée, le nombre d'instructions élémentaires exécutées pour calculer `PGCD(i,j)` est en O(I).

Autre exemple: PGCD - Euclide (2)

Il faut maintenant ***déterminer I***

Pour cela, récrivons `PGCD(i,j)` de la façon suivante:

```
PGCD(i,j) {
   if (j==0) return (i);
   else {
      if ((i%j) == 0) return (j);
      else return(PGCD((i%j), j % (i%j)));
}  }
```

On voit que le calcul de `PGCD(i,j)` entraîne celui de `PGCD((i%j),j%(i%j))`
On a par ailleurs `(x%y)` ≤ `(x/2)`;[4] le nombre maximal d'appels à PGCD() est donc en O($\log_2$`i`).

On obtient donc que le nombre maximal d'instructions élémentaires nécessaires pour calculer `PGCD(i,j)` est en O($\log_2$`i`)...

... comme $\log_2 i$ est également une mesure possible pour la taille des entrées du problème, on obtient finalement que l'algorithme d'Euclide est **linéaire**.

4. En effet, si y ≤ (x/2), alors comme (x% y) < y, on à (x% y) < (x/2).
De plus, si (x/2) ≤ y, alors 1 ≤ (x/y) < 2, d'où (x÷y) = 1 et (x% y) = x-(y·(x÷y)) = (x/2) puisque (x-y) < x-(x/2).

Programmation dynamique (1)

La ***programmation dynamique*** est
un schéma de résolution algorithmique permettant de traiter
des problèmes ayant une **structure séquentielle**.

Plus précisément, on pourra utiliser la programmation dynamique pour résoudre un problème P, si celui-ci peut être plongé dans une famille de problèmes f telle que

- $P \in f$
- $\forall$ P, il existe un ensemble **fini** $\{P_1,\ldots P_k\}$ d'éléments de f t.q. :
  - $\forall i$, P_i est un **sous-problème** de P,
  - et **une solution** pour P **peut être dérivée** à partir des solutions pour $P_1,\ldots P_k$

On dira que P' est un sous-problème de P
si c'est un problème **de même nature**, mais appliqué
à des données d'entrée **de tailles plus petites**.

En termes plus simples, un problème a de bonnes chances d'être soluble par une approche de type programmation dynamique s'il peut être décomposé en un nombre fini de sous-problèmes de même type, dont les solutions permettent de construire une solution au problème initial.

Programmation dynamique (2)

Exemple: Calcul de C_n^p $(n \geq p \geq 0)$

Par définition de C_n^p on a, $\forall\ n \geq p \geq 0$, $C_n^p = \begin{cases} 1 \text{ si } p=0 \text{ ou } p=n \\ C_{n-1}^{p-1} + C_{n-1}^{p} \text{ sinon} \end{cases}$

Soit alors P(n,p) le problème du calcul de C_n^p.

P(n,p) peut être plongé dans la famille $f(n,p) = \left\{ P(i,j) \;\middle|\; \begin{array}{c} 0 \leq i \leq n \\ 0 \leq j \leq p \\ j \leq i \end{array} \right.$

et on a bien:

$P(n,p) \in f(n,p)$

et P(n,p) peut être résolu à partir de P(n-1, p-1) et P(n-1,p), pour n > p > 0, P(n,p=n) et P(n,0) sont par ailleurs directement solubles: P(n,n) = P(n,0) = 1

L'algorithme de résolution de P(n,p) est alors:

```
Calcul(n,p) {
   if ((n==p) || (p==0)) return(1);
   else return( Calcul(n-1,p-1) + Calcul(n-1,p) );}
```

Programmation dynamique (3)

La solution par programmation dynamique proposée n'est cependant pas très efficace, puisque les valeurs des C_j^i sont calculées plusieurs fois lors des appels récursifs.

Pour rendre l'algorithme efficace, il faut donc **mémoriser ces valeurs intermédiaires**.[5]

Une solution simple est de créer le tableau triangulaire nécessaire, et de le remplir au fur et à mesure.

L'algorithme est alors:

```
Calcul(n,p) {
   Table t = creeTable(n,p);    // le tableau est créé et ses cellules
                                // sont initialisées à 1 sur la diag.
   return Calcul2(n,p,&t);      // et la 1ère colonne, et 0 ailleurs
}

Calcul2(n,p,t) {
   if (t[n][p]==0)
      t[n][p]=Calcul2(n-1,p-1,t)+Calcul2(n-1,p,t);
   return (t[n][p]);
}
```

n \ p	0	1	2	3	...	p-1	p
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
⋮	⋮						
n-1	1					... C_{n-1}^{p-1}	... C_{n-1}^{p}
n	1						... C_n^p

5. Notez, et c'est une règle souvent vérifiée, qu'une meilleure complexité en temps a été obtenue au prix d'une complexité en espace plus importante. C'est l'une des grandes loi de l'informatique: «une réduction du temps de traitement se paye souvent par une augmentation de la consommation mémoire, et inversément».