INF3105 – Analyse et Complexité algorithmique

Éric Beaudry

Université du Québec à Montréal (UQAM)

Été 2024

Sommaire

Complexité algorithmique

- Complexité $\neq$ difficulté à comprendre un algorithme.
- Complexité = quantité des ressources (temps processeur, mémoire) requises.
- Plus un programme nécessite de *ressources*, plus il est *complexe*.
- N'est pas le sujet principal d'INF3105.
- Base nécessaire pour évaluer, comparer et choisir des structures de données.
- Cours :
  - INF1120, INF1132 et INF2120 : Aperçu de la complexité.
  - INF3105 : Rappel des notions de base + Analyse des structures fondamentales et algorithmes reliés.
  - INF5130 (Algorithmique) : cours dédié au sujet.

Éléments à évaluer à propos des algorithmes

1. Complexité temporelle : temps d'exécution (temps processeur).
2. Complexité spatiale : quantité de mémoire.

Facteurs affectant le temps d'exécution / quantité de mémoire d'un programme

- Principal facteur :
  - **La taille du problème**.
  - Exemples : trier n nombres ; décompresser une image de $w \times h$ pixels ; inverser une matrice carrée de $n \times n$; etc.
- Facteurs secondaires :
  - Matériel (processeur, mémoire, etc.).
  - Langage de programmation, compilateur, configuration du compilateur, etc.
  - Qualité de l'implémentation de l'algorithme à évaluer.
  - Système d'exploitation.
  - Etc.

Expression à l'aide d'une fonction

- Le temps d'exécution (la complexité temporelle) et la quantité de mémoire requise (complexité spatiale) peuvent s'exprimer à l'aide d'une **fonction** f ayant pour paramètres les **principaux facteurs**.
- Exemple : $f(n)$ où n est la taille du problème.
- Permet d'estimer (prédire) le temps d'exécution d'un programme (algorithme) pour une entrée donnée.
- Quand on s'intéresse aux algorithmes (partie théorique) et non aux programmes (partie implémentation), on fait généralement abstraction des facteurs secondaires.
- Les facteurs secondaires deviennent pertinents quand on s'intéresse à un système précis.

Méthodes d'analyse

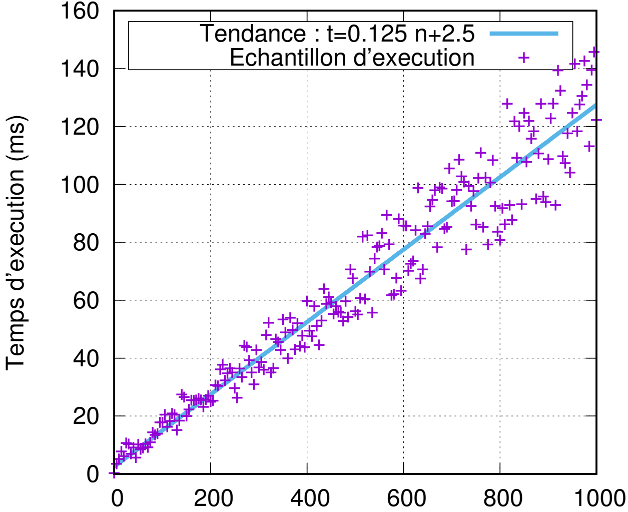
Comment trouver ou estimer la fonction $f(\cdots)$:

1. Analyse empirique.
2. Analyse asymptotique.

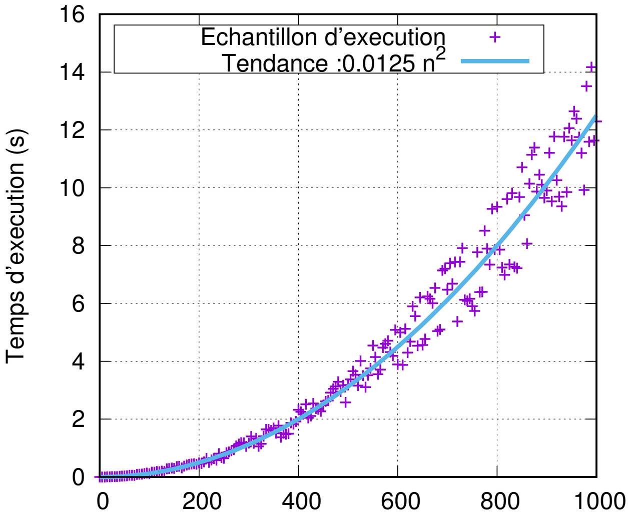
Analyse empirique

1. Écrire un programme qui implémente un algorithme.
2. Écrire des problèmes test de différentes tailles.
3. Exécuter le programme sur les problèmes et mesurer le temps.
   Idéalement le temps CPU, mais peut être le temps réel.
   Exemples :
   - chronomètre de votre montre (pas le meilleur choix).
   - commande `time` sous Linux, Unix, etc. ;
   - fonction `getrusage()` en C sous Linux, Unix, etc. ;
   - fonction `System.getCurrentTime()` en Java ;
4. Tracer un graphique.
5. Extrapoler une relation $f : n \rightarrow temps$.

Analyse empirique



Analyse empirique



Avantages / Inconvénients

Avantages

- Méthode simple.
- Si les tests sont représentatifs, alors les mesures observées sont représentatives.

Inconvénients

- Généralement difficile de couvrir tous les cas possibles.
- Dans ce cas : estimations imprécises.
- Difficile de garantir le pire cas.

Neutre (parfois un avantage, parfois un inconvénient)

- Considère implicitement les facteurs secondaires.

Notation grand O

- Généralement, on ne s'intéresse qu'à un **ordre de grandeur**.
- Notation : $O(g(n))$ où on remplace $g(n)$ par une formule contenant n (ou d'autres variables).
- Formellement, $O(g(n))$ est un ensemble de fonctions
- $O(g(n)) = \{f(n) | \exists k, c, f(n) \leq c \cdot g(n), \forall n \geq k\}$
- Interprétation : $O(g(n))$ contient toutes les fonctions $f(n)$ qui ne croissent pas asymptotiquement plus rapidement que $g(n)$.
- Exemples :
  - $f_1(n) = n$
  - $f_2(n) = 2n + 4$
  - $f_3(n) = \frac{n}{3}$
  - $f_4(n) = n^2 + 20n + 199$
  - $f_1 \in O(n)$, $f_2 \in O(n)$, $f_3 \in O(n)$, $f_4 \in O(n^2)$

Simplification en INF3105 (absence de Ω et Θ)

- INF3105 : on exprimera la complexité avec une notation simplifiée de O.
- INF5130 abordera aussi les symboles O, Ω et Θ :
  - $O(g(n))$: ensemble des fonctions $f(n)$ tel que $f(n) \leq c \cdot g(n)$ et ... (borne supérieure)
  - $\Omega(g(n))$: ensemble des fonctions $f(n)$ tel que $f(n) \geq c \cdot g(n)$ et ... (borne inférieure)
  - $\Theta(g(n))$: ensemble des fonctions $f(n)$ tel que $f(n) \in O(g(n))$ et $f(n) \in \Omega(g(n))$

Simplification d'expression en notation grand O

- Question : si $f(n) = 2n$, alors $f(n)$ est-elle dans $O(2n)$?.
- Réponse : oui, car $f(n) \in O(2n)$. Mais, $f(n)$ est aussi dans $O(n)$.
- Question : $O(2n) = O(n)$? Réponse : oui.
- Il est préférable d'écrire $O(n)$ plutôt que $O(2n)$, car il s'agit de l'expression la plus simple.
- Analogie : avec les fractions, nous écrivons rarement $\frac{2}{4}$; nous écrivons plutôt $\frac{1}{2}$, car il s'agit de l'expression la plus simple.
- Exemple :
  - $f(n) = 7n^4 + 5n^3 + 2n^2 + 9n + 19$
  - À quel ordre de grandeur appartient $f(n)$?
  - On garde le terme ayant le degré le plus élevé du polynome : $7n^4$.
  - On élimine la constante 7 devant n^4.
  - Donc : $f(n) \in O(n^4)$

Exemples de simplifications en notation grand O

	Fonction	Ordre de grandeur
1	n	$O(n)$
2	$2n+3$	$O(n)$
3	$2n^2+8n-3$	$O(n^2)$
4	$\frac{1}{2}n^3+8n^2+3n+5$	$O(n^3)$
5	$\log_2 n$	$O(\log n)$
6	$\log_{10} n$	$O(\log n)$
7	$7n+3\log_2 n$	$O(n)$
8	$7n\log_2 n+9n$	$O(n\log n)$
9	$n^2+2n\log_{10}\frac{n}{2}+3n$	$O(n^2)$
10	2^n+2n^4	$O(2^n)$
11	$3n!$	$O(n!)$

Classes de complexité

Ordre	Complexité	Exemples
$O(1)$	Temps constant	Un accès aléatoire, un calcul arithmétique, etc.
$O(\log n)$	Logarithmique	Recherche dichotomique (binaire) dans un tableau trié.
$O(n)$	Linéaire	Itérer sur les éléments d'un tableau ou d'une liste.
$O(n \log n)$	« n log n »	Tri de fusion et de monceau. Tri rapide (excepté le pire cas).
$O(n^2)$	Quadratique	Parcours d'un tableau 2 dimensions. Tri de sélection.
$O(n^3)$	Cubique	Multiplication matricielle naïve.
$O(b^n)$	Exponentiel	Problèmes de planification. ($b \geq 2$)
$O(n!)$	Factoriel	Problèmes d'ordonnancement. Problème du voyageur de commerce.

Méthode d'analyse

- **Compter (dénombrer) le nombre d'opérations en fonction de la taille du problème**.
- On ne fait pas de différence entre la nature des opérations, même si elles ne prennent pas le même temps en pratique.
- On fait abstraction des facteurs secondaires (CPU, type d'opérations, langage de programmation, etc.).
    - Les facteurs secondaires sont (généralement) indépendant de la taille du problème.
    - Les facteurs secondaires se résument (généralement) à une constante.
- Rappel : on s'intéresse en premier lieu à l'ordre de grandeur.

Quoi analyser ?

- **Cas moyen**. Moyenne de toutes les entrées possibles.
- **Pire cas**. Pire entrée possible.
- **Analyse amortie** : le temps moyen d'une opération répétée plusieurs fois dans le cadre d'une autre opération de plus haut niveau.

Exemple 1

moyenne1.cpp

```cpp
int main(){
    int n;
    double somme = 0;
    cin >> n;
    for(int i=0;i<n;i++){
        double x;
        cin >> x;
        somme += x;
    }
    cout << "moyenne : " << (somme / n);
}
```

Exemple 2

moyenne2.cpp

```
int main(){
    int n;
    double somme = 0;
    cin >> n;
    double* tab = new double[n];
    for(int i=0;i<n;i++)
        cin >> tab[i];
    for(int i=0;i<n;i++)
        somme += tab[i];
    cout << "moyenne : " << (somme / n);
    delete[] tab;
}
```

Exemple 3

```
int main(){
    int n;
    cin >> n;
    bool doublons = false;
    string* tab = new string[n];
    for(int i=0;i<n;i++) cin >> tab[i];
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            if(i!=j)
                doublons |= tab[i]==tab[j]; //if(tab[i]==tab[j]) doublons=true;
    delete[] tab;
}
```

Exemple 4

exemple4.cpp

```
int main(){
    int n;
    cin >> n;
    bool doublons = false;
    string* tab = new string[n];
    for(int i=0;i<n;i++) cin >> tab[i];
    for(int i=0;i<n && !doublons;i++)
        for(int j=i+1;j<n;j++)
            doublons |= tab[i]==tab[j];
    delete[] tab;
}
```

Exemple d'analyse amortie (1) : contexte

```
class SeqN{
  public:
    ...
    bool contient1(int nombre);
    bool contient2(int nombre);
  private:
    int* tableau; // hypothèse: tableau trié
    int taille    // taille du tableau
    int position; // dernière position utilisée par contient2
};
int compte(int n, int* t1, int* t2){
  int k=0;
  SeqN seqn(t1,n);
  for(int i=0;i<n;i++)
    if(seqn.contientX(t2[i]) // remplacez X par 1 ou 2
      k++;
  return k;
}
int main(){
  int n=8;
  int t1[n] = {5, 6, 7, 8, 10, 20, 40, 48};
  int t2[n] = {0, 1, 2, 3, 20, 40, 41, 48};
  return compte(n, t1, t2);
}
```

Exemple d'analyse amortie (2) : version naïve

```
bool SeqN::contient1(int nombre){
  for(int i=0;i<taille;i++)
    if(tableau[i]==nombre)
      return true;
  return false;
}
int compte(int n, int* t1, int* t2){
  int compte=0;
  SeqN seqn(t1);
  for(int i=0;i<n;i++)
    if(seqn.contient1(t2[i]))
      compte++;
  return compte;
}
```

Exemple d’analyse amortie (3) : version améliorée

```
bool SeqN::contient2(int nombre){
  if(nombre < tableau[position]) position=0;
  while(position<taille−1 && tableau[position]<nombre)
    position++;
  return tableau[position]==nombre;
}
int compte(int n, int* t1, int* t2){
  int compte=0;
  SeqN seqn(t1);
  for(int i=0;i<n;i++)
    if(seqn.contient2(t2[i]))
      compte++;
  return compte;
}
```

Tri de sélection

```
TriSelection(a[0 : n − 1])
    pour i = 0, . . . , n − 1
        k ← i
        pour j = k + 1, . . . , n − 1
            si a[j] < a[k]
                k ← j
        Échanger(a[i], a[k])
```

```
TriFusion(a[0 : n−1])
   si n ≤ 1 retourner
   m ← ⌊n/2⌋

   TriFusion(a[0 : m−1])
   TriFusion(a[m : n−1])

   créer b[0 : n−1]
   i ← 0
   j ← m
   k ← 0
   Tant que  i ≤ m et j < n
      b[k++] ← a[j] < a[i] ? a[j++] : a[i++]
   Tant que  i < m
      b[k++] ← a[i++]
   Tant que  j < n
      b[k++] ← a[j++]
   a ← b
```

```cpp
template <class T> void tri_rapide(T* tab, int n){
    if(n<=1) return;
    int p = 0; // Choisir un pivot: plusieurs possibilités dont le premier (p=0)
    //au millieu: int p = (n-1)/2;  au hasard: int p = random(n);
    swap(tab[0],tab[p]);
    //Diviser le vecteur en deux
    int k = 0;
    for(int i = 1;i<n;i++)
        if(tab[i] < tab[0])
            swap(tab[++k],tab[i]);
    swap(tab[0],tab[k]);    //On obtient tab[0:k-1] < tab[k]<= tab[k+1:n-1]
    //Appels recursifs
    tri_rapide(tab, k);
    tri_rapide(tab+k+1, n-k-1);
}
```

```
int main(){
    int n=0, // n: le nombre de mots lus dans le texte en entrée
        m=0; // m: le nombre d'entrée originale := synonyme dans le dictionnaire de synonymes
    ifstream fsynonymes("synonymes.txt"); /**** Lecture d'entrées dans un dictionnaire sous forme de fichier texte ***/
    fsynonymes >> m; // nombre de synonymes dans le fichiers
    string *originaux = new string[m], *synonymes = new string[m];
    for(int i=0;i<m;i++)
        fsynonymes >> originaux[i] >> synonymes[i];
    while(cin){  /*** Lecture d'un texte depuis l'entrée standard ***/
        string mot;
        cin >> mot;
        for(int i=0;i<m;i++)
            if(mot==originaux[i]){
                mot = synonymes[i];
                break;
            }
        cout << mot << " ";
        n++;
    }
    cout << endl;
}
```

Complexité du programme précédent ?

- Quelle est la taille du problème ?
- La taille du problème peut se définir avec 2 variables :
  - m : le nombre d'entrée original := synonyme dans le dictionnaire de synonymes.
  - n : le nombre de mots lus dans le texte en entrée.
- On ne connaît pas à l'avance les valeurs de n et m.
- Possibilités :
  - $m \approx n$.
  - $m < n$ ou même $m \ll n$.
  - $m > n$ ou même $m \gg n$.
- Complexité du programme précédent : $O(mn)$.