TD1.1 – Analyse d'algorithmes, calculs de coûts

Objectifs

À la fin de cette séance, vous devriez être capable de :
— évaluer le coût d'un algorithme simple dans le pire et dans le meilleur des cas ;
— comparer des algorithmes selon leur complexité ;
— évaluer la qualité d'un algorithme selon sa complexité.

Exercice 1 : Itérations emboîtées (30 min)

Compter le nombre d'opérations *Schtroumpfer* exécutées par chacun des algorithmes suivants.

1.
```
(1) pour i = 1 à n faire
(2)     pour j = 1 à n faire
(3)         Schtroumpfer()
```
2.
```
(1) pour i = 1 à n faire
(2)     pour j = 1 à i faire
(3)         Schtroumpfer()
```
3.
```
(1) pour i de 5 à n-5 faire
(2)     pour j de i-5 à i+5 faire
(3)         Schtroumpfer()
```
4.
```
(1) pour i = 1 à n faire
(2)     pour j = 1 à i faire
(3)         pour k = 1 à j faire
(4)             Schtroumpfer()
```
5. Plus généralement, que pouvez-vous dire de la complexité d'un algorithme en observant le nombre de boucles emboîtées ?

Correction de l'exercice 1

Pour tous ces algorithmes la ligne Schtroumpfer() effectue exactement un schtroumpfage (et c'est la seule).

1. La boucle (2) s'exécute n fois et elle effectue donc n schtroumpfages.
   La boucle (1) s'exécute n fois et elle effectue donc $n \times n = n^2$ schtroumpfages.
2. La boucle (2) s'exécute i fois et elle effectue donc i schtroumpfages.
   La boucle (1) s'exécute n fois, pour i allant de 1 à n. Elle effectue donc $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ schtroumpfages.
3. Il y a ici un "piège" car le nombre d'itérations de la boucle interne ne dépend pas de i.
   La boucle (2) s'exécute $(i+5)-(i-5)+1 = 11$ fois et elle effectue donc 11 schtroumpfages.
   La boucle (1) s'exécute $(n-5)-4 = n-9$ fois et elle effectue donc $11(n-9) = 11n-99$ schtroumpfages.
4. La boucle (3) s'exécute j fois et elle effectue donc j schtroumpfages.
   La boucle (2) s'exécute i fois, pour j allant de 1 à i. Elle effectue donc $\sum_{j=1}^{i} j = \frac{i(i+1)}{2} = \frac{i^2}{2} + \frac{i}{2}$ schtroumpfages.
   La boucle (1) s'exécute n fois, pour i allant de 1 à n. Elle effectue donc $\sum_{i=1}^{n} \frac{i^2}{2} + \frac{i}{2} = \frac{1}{2}\sum_{i=1}^{n} i^2 + \frac{1}{2}\sum_{i=1}^{n} i = \frac{1}{2}\frac{n(n+1)}{2} + \frac{1}{2}\frac{n(n+1)(2n+1)}{6}$ schtroumpfages.
   Notons que cette dernière expression est en $\mathcal{O}(n^3)$.
5. La complexité sera en général un polynôme dont le degré est égal au nombre maximal de boucles emboîtées, **à condition que l'intervalle de variation de chaque indice soit proportionnel à celui d'un des indices supérieurs**.

Exercice 2 : Recherche séquentielle (30 min)

On étudie un algorithme de recherche séquentielle dans une table. On se place dans le cas où il n'y a pas d'hypothèse sur le fait que la table est ordonnée ni sur la présence de l'élément cherché dans la table.

1. Spécifier et écrire proprement un tel algorithme.
2. Il s'agit maintenant d'évaluer le nombre de comparaisons effectuées lors de la recherche. Déterminer les cas favorables et défavorables et les nombres de comparaisons correspondants. Démontrer qu'ils correspondent bien respectivement au minimum et au maximum du coût de votre algorithme.

Correction de l'exercice 2

1. Les indices du tableau vont de 0 à $n-1$

```
RECH_SEQ(e, T, n)
Data : l'élément e à rechercher, un tableau T de taille n
Result : le premier indice i tel que T[i] = e, ou −1 s'il est absent
i := 0 ;
while i < n et puis T[i] ≠ e do
    i := i + 1
if i < n then
    return i
else
    return −1
```

2. Les pires cas sont faciles à exhiber : ce sont ceux où la boucle effectue un maximum d'itérations, autrement dit lorsque i va de 0 à n. Cela se produit si e est absent de T ou en dernière position, d'où $C^{max}_{RECH\_SEQ}(n) = n$.
   Le cas favorable est celui où $T[0] = e$, et alors $C^{min}_{RECH\_SEQ}(n) = 1$. On se convainc facilement que le coût ne peut être inférieur : la première comparaison $0 < n$ est forcément vraie et donc on compare toujours $T[0]$ avec e quel que soit le tableau T.

Exercice 3 : Valeurs numériques et ordres de grandeurs (30 min)

On suppose qu'on travaille sur une machine capable d'effectuer environ un milliard d'opérations par seconde.

Calculer (**sans calculatrice**) le temps nécessaire approximatif pour exécuter des programmes dont les coûts sont donnés ci-dessous, avec des données de différentes tailles en entrée :

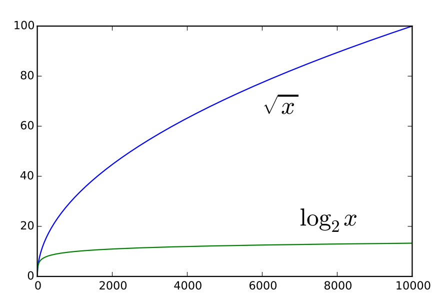
↓ Coût de l'algorithme \ Taille des données →	1 000	1 000 000	1 000 000 000
n			
$n \log_2 n$			
$n + 1\,000\,000$			
$\frac{n^2}{1\,000} + 1\,000n$			

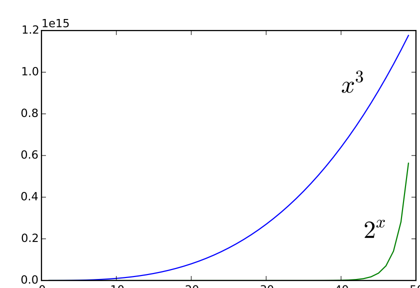
1. Lesquels de ces algorithmes sont utilisables :
   — à chaque chargement d'une page web ?
   — à chaque démarrage d'une machine ?
   — pour produire les plans d'une usine ?
   Quelle(s) conclusion(s) plus générale(s) en tirez-vous sur les ordres de grandeurs respectifs de ces coûts ?
2. Tracer sur une même figure les allures des courbes des fonctions $\log_2(n)$ et $\sqrt{n}$ sur une échelle assez grande ($n = 10\,000$ par exemple).
   Même consigne pour $10^{10} \times n^3$ et 2^n, avec n compris entre 0 et 50.

Correction de l'exercice 3

↓ Coût de l'algorithme \ Taille des données →	1 000	1 000 000	1 000 000 000
n	10^{-6} s	10^{-3} s	1 s
$n \log_2 n$	10^{-5} s	$1/50$ s	30 s
$n + 1\,000\,000$	10^{-3} s	2×10^{-3} s	1 s
$\frac{n^2}{1\,000} + 1\,000n$	10^{-3} s	2 s	11 jours et demi

1. — n et $n + 1\,000\,000$, à la limite la version $n \log_2 n$ si pas trop de données à traiter
   — Tous sauf $n^2/1000$
   — Tous (on ne construit pas une usine tous les jours !), à condition que $n = 10^9$ soit considéré comme un volume de données raisonnable pour ce problème.
   — n et $n + 1\,000\,000$ sont équivalents.
   — $n \log_2$ est presque du même ordre sauf pour de très grandes données.
   — n^2 est nettement au-dessus, même avec une constante faible.
2. Il faut voir ici que les allures de courbes apprises au lycée ne sont plus vraiment valables pour n grand, et en particulier que $\log_2$ et 2^n ne présentent pas de branche pseudo-parabolique comme on a tendance parfois à le croire.





Exercice 4 :

On considère l'algorithme suivant :

```
(1)  a := T[0]
(2)  pour i = 1 à n-1 faire
(3)          si T[i] > a faire
(4)              traiter T[i]
(5)              a := T[i]
```

1. Que calcule-t-il ? Expliciter les données, le résultat.
2. Expliciter le modèle de coût. Quel est son coût minimal, son coût maximal ?
3. Peut-on espérer écrire un algorithme qui calcule le maximum d'un tableau avec strictement moins de comparaisons que celui-ci ?

Correction de l'exercice 4

1. Il calcule le maximum d'un tableau. Les données sont T et n, le résultat est a.
2. Considérons que l'appel à traiter est l'opération coûteuse et négligeons le reste.
   Selon le résultat du test, le coût de l'instruction conditionnelle (3) est compris entre 0 et 1.
   Par conséquent, le coût de la boucle (2) est compris entre 0 et $n-1$ et on peut exhiber des cas favorable et défavorable pour lesquels ces bornes sont atteintes.
   Donc $C^{min}(n) = 0$ et $C^{max}(n) = n - 1$.
3. Supposons qu'un algorithme A effectue strictement moins de comparaisons, donc n-2 au maximum.
   Alors quelles que soient les comparaisons effectuées, il existe deux sous-ensembles T_1 et T_2 de T tels qu'aucun élément de T_1 ne soit comparé avec un élément de T_2.
   Si A choisit un élément de T_1 comme maximum, on peut construire une donnée telle que A choisisse le même élément mais que le maximum réel soit dans T_2.
   Un tel algorithme A ne peut donc pas être correct.