ALGORITHMIQUE : TRIS NAÏFS

INTRODUCTION DES NOTATIONS

Objectifs. — On se fixe les objectifs suivant pour ce cours
- — compter le nombre d'opérations effectué par des algorithmes ;
- — créer des programmes conçus pour toutes les valeurs d'un certain type.

L'évaluation de complexité est *asymptotique*. Le nombre d'opérations est exprimé en fonction de la « taille » [1§] des entrées.

> Définition 0.0.0.1 (Notation O()). —
> Soient $f, g : \mathbf{N} \to \mathbf{R}^+$. On dit que $f = \mathrm{O}(g)$ si :
> $$\exists C \in \mathbf{R}^+, \exists n_0 \in \mathbf{N}, \forall n \geq n_0,\ f(n) \leq C \cdot g(n).$$

Dans la suite, $f : \mathbf{N} \to \mathbf{R}^+$ sera l'application qui à une entrée de taille n associe le nombre d'opérations nécessaires.

> Définition 0.0.0.2 (Notation Ω()). —
> Soient $f, g : \mathbf{N} \to \mathbf{R}^+$. On dit que $f = \Omega(g)$ si :
> $$\exists C \in \mathbf{R}^+, \exists n_0 \in \mathbf{N}, \forall n \geq n_0,\ f(n) \geq Cg(n).$$

> Définition 0.0.0.3 (Notation Θ()). —
> Soient $f, g : \mathbf{N} \to \mathbf{R}^+$. On dit que $f = \Theta(g)$ si :
> $$f = \mathrm{O}(g) \text{ et } f = \Omega(g).$$

1§. Par exemple : pour un entier $x \in \mathbf{N}$ ça peut être la longueur de son expression en décimal ; pour une liste cela peut être le nombre d'éléments de la liste, si l est de longueur n alors sa taille peut être : $\text{taille}(l) = |l| = \sum_{i=0}^{n-1} \text{taille}(l[i])$

Exercice. — Montrer que $f = \mathrm{O}(g)$ lorsque :

1. $f(n) = n^d$ et $g(n) = n^{d+i}, i \geq 0$;
2. $f(n) = \log n$ et $g(n) = n^\varepsilon,\ \varepsilon > 0$;
3. $f(n) = n^k$ et $g(n) = r^n$ avec $r > 1$ et $k \in \mathbf{N}$.

Définition 0.0.0.4 (Opérations dans les algorithmes). —
Voici les éléments comptés :
- le nombre d'affectations ;
- le nombre de comparaisons ;
- le nombre de divisions :
- ...

Exemple. — Le nombre de divisions effectuées par l'algorithme d'Euclide pour a, b (avec $a > b$) est une fonction de b : $\Theta(\log b)$.

1. PERMUTATIONS

Objectifs. — Faire un tour d'horizon des algorithmes de tris.

Opérations. — Les opérations intéressantes pour les tris :
- comparaisons ;
- affectation et échanges de variables.

Proposition 1.0.0.1 ($\mathbf{N}$ est totalement ordonné). —
Les propriétés suivantes indiquent que $\mathbf{N}$ est totalement ordonné avec $<$:

$$\begin{cases} \forall x \neq y,\, x < y \text{ ou } y < x, \\ \forall x, y, z,\, x < y \text{ et } y < z \implies < z. \end{cases}$$

Définition 1.0.0.5 (Trier). —
Trier un tableau ou une liste d'entiers : T de longueur $n \in \mathbf{N}$ c'est trouver une permutation $\sigma \in S_n$ telle que :

$$\forall i \leq n - 1,\ T[\sigma(i)] \leq T[\sigma(i+1)].$$

Exemple. — Pour $T = (10, 9, 7, 6, 8)$ on a :

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 0 & 2 \end{pmatrix}.$$

Lemme 1.0.0.1. —
Pour tout $n \in \mathbf{N}^*$, toute permutation de S_n peut s'écrire comme un produit de transpositions.

2. TRI À BULLE

2.1. Principe

Le principe de cet algorithme est de faire « remonter » les valeurs les plus grandes vers la fin de la liste jusqu'à ce que le tableau soit *trié*.

> DÉFINITION 2.1.0.6. —
> Soit T un tableau.
> On note sa longueur longueur$(T) = |T| = n$. On indice le tableau de 0 à $n-1$.
> Un tableau est *trié* si :
> $$\forall i \in \{0, \ldots, n-2\}, \; T[i] \leq T[i+1].$$

On peut écrire un premier algorithme pour tester si le tableau est bien trié.

```
function tableauTrie(T)
        n = longueur(T)
        for i from 0 to n-2 do
                if T[i] > T[i+1] then return false
```

2.2. Principe détaillé du tri Bulle (Bubble sort)

On fait dans l'ordre :
— un balayage successif de T de gauche à droite ;
— au k-ième balayage :
  — on parcourt les $n-k$ valeurs les plus à gauche ;
  — si une position j est telle que $T[j] > T[j+1]$ alors on échange $T[j]$ et $T[j+1]$.

Écriture de l'algorithme. — Une première écriture donne :

```
function triBulle(T)
        n = longueur(T)
        for i from n-1 to 0 do
                for j from 0 to i-1 do
                        if T[j] > T[j+1] then
                                T[j],T[j+1] = T[j+1],T[j]
        return T
```

En tenant compte du fait que si un balayage entier est effectué sans modification alors c'est que le tableau est trié, on obtient :

```
function triBullePlus(T)
        n = longueur(T)
        i = n-1
        repeat
```

```
            trie = true
            for j from 0 to i-1 do
                if T[j] > T[j+1] then
                    T[j],T[j+1] = T[j+1],T[j]
                    trie = false
            i = i-1
    until trie = true
    return T
```

2.3. Preuve de correction

On va montrer l'existence d'invariants.

Proposition 2.3.0.2 (Invariants). —
Soit $k \in \{0, \ldots, n-1\}$. Si $i = k$ alors :

1. $T[k+1] \leq T[k+2] \leq \ldots \leq T[n-1]$;
2. $\forall i < k+1, T[i] \leq T[k+1]$.

Démonstration 2.3.0.1 (Par induction/récurrence sur $n-1-k$)
Si $n-1-k = 0$ alors c'est bien vérifié.
Supposons l itérations de l'algorithme et que la propriété est vraie pour $n-1-k = l$.
Dans ce cas, à l'itération $l+1$:

— La boucle interne met en position $n-1-l$ la plus grande des valeurs v entre les positions 0 et $n-1-l$ de T.

— Par hypothèse de récurrence, $v \leq T[k+1] \leq T[k+2] \leq \ldots \leq T[n-1]$, on a donc :

$$T[k] \leq T[k+1] \leq \ldots \leq T[n-1] \text{ et } \forall i < k, T[i] \leq T[k].$$

2.4. Analyse de complexité

On note A un algorithme, T une entrée de A (ici un tableau) et $C_A(T)$ le nombre d'opérations [2§] que l'algorithme A effectue sur l'entrée T.

Définition 2.4.0.7. —
La *complexité de A dans le pire des cas* est la fonction :

$$\mathcal{C}_A : \begin{cases} \mathbf{N} \to \mathbf{N} \\ n \mapsto \max\limits_{T:|T|=n} C_A(T) \end{cases}.$$

C'est-à-dire le nombre d'opération maximal sur une entrée de taille n.

2§. Ici : échanges, comparaisons et affectations.

Remarque. — Soit $f : \mathbf{N} \to \mathbf{N}$:

— La complexité de A est en $\mathrm{O}(f(n))$ si

$$\mathcal{C}_A(n) \leq \mathrm{O}(f(n)).$$

— La complexité de A est en $\Omega(f(n))$ s'il existe n_0, T de taille $n \geq n_0$ et $c \in \mathbf{R}$ tel que

$$\mathcal{C}_A(T) \geq c \cdot f(n).$$

Tri Bulle. — On va essayer d'établir le nombre de comparaisons $C(n)$:

— pour n balayages :
  — au k-ième balayage on effectue $n - k - 1$ comparaisons.

On a alors :

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} \frac{n(n-1)}{2}.$$

Ainsi $C(n)$ est en $\mathrm{O}(n^2)$ et en $\Omega(n^2)$. Le nombre d'échanges, $E(n)$ est en $\Omega(n^2)$ avec comme pire des cas un tableau trié à l'envers.

3. TRI PAR SÉLECTION

3.1. Principe

Le principe du tri par sélection est de chercher le maximum et de le placer à la fin. On itère avec T entre les indices 0 et $n-2$ (avec $n = |T|$).

L'algorithme qui trouve le maximum :

```
function posMax(T,p,f)
        k = p
        for i from p+1 to f do
                if T[i] > T[k] then
                        k = i
        return k
```

L'algorithme de tri en version itérative :

```
function triSelectionIter(T,p,f)
        for i from f to p+1 do
                j = posMax(T,p,i)
                T[i],T[j] = T[j],T[i]
```

Et en version récursive :

```
function triSelectionRec(T,p,f)
        if p < f then
                j = posMax(T,p,f)
                T[f],T[j] = T[j],T[f]
                triSelectionRec(T,p,f-1)
```

3.2. Analyse de complexité (pour la version récursive)

On compte le nombre d'échanges, $E(n)$, les affectations, $A(n)$, et les comparaisons, $C(n)$.

Par récurrence :

— Si $|T| = 0, 1$ alors $C(0) = E(0) = A(0) = 0$.
— Si $|T| = 2$ un appel à posMax, $C(1) = A(1) = 1$ et $E(1) = 1$.
— Si $|T| = n + 1$ alors :

1. un appel à posMax avec sur T : n comparaisons et n affectations (au plus) ;
2. un échange ;
3. un appel à triSelectRec sur le sous-tableau de taille $n - 1$.

On a donc $C(n + 1) = n + C(n)$, $A(n + 1) \leq n + C(n)$ et $E(n + 1) \leq 1 + E(n)$. Au final $C(n)$ et $A(n)$ sont en $\mathrm{O}(n^2)$ et en $\Omega(n^2)$ (pour un tableau trié en ordre décroissant) et $E(n)$ est en $\mathrm{O}(n)$ et en $\Omega(n)$.

4. RECHERCHE DANS UN TABLEAU TRIÉ

Rechercher est une activité essentielle en informatique. L'objectif est de le faire à moindre coût. On effectue un tri pour pouvoir rechercher efficacement.

4.1. Recherche naïve

Elle consiste à essayer toutes les possibilités :

```
function rechercheBinaire(T,m,n,x)
        j = m
        while j < n+1 do
                if T[j] = x then return j
                else j = j+1
```

4.2. Recherche dichotomique

Principe. — On compare l'élément x à chercher avec la valeur du milieu du tableau. En fonction de la réponse on dirige la recherche suivante vers l'une des deux moitiés du tableau.

```
function rechercheBinaire(T,l,u,x)
        if u < l then return
        m = int((l+u)/2)
        if T[m] = x then return m
        if T[m] > x then
                return rechercheBinaire(T,l,m-1,x)
        return rechercheBinaire(T,m+1,u,x)
```

4.3. Complexité de la recherche dichotomique

On va compter le nombre de comparaisons, $C(n)$.

— $C(0) = 0$;
— $C(1) = 1$ (car on vérifie si c'est x ou non) ;
— $C(2) = 2$;
— si $u > l$, $m - l \leq u - m \leq m - l + 1$ et alors $1 + \max(m - l, u - m) \leq \frac{u-l+1}{2}$ et donc $C(n) \leq 1 + C(\lfloor \frac{n}{2} \rfloor)$.

Lemme 4.3.0.2. —
La recherche dichotomique nécessite au plus

$$C(n) \leq \lfloor \log_2(n) + 1 \rfloor \leq \log_2 n + 1$$

Preuve en exercice. — Par récurrence sur $k = \lfloor \log_2 n \rfloor$. Indication :

$$C(n) \leq 1 + C(\lfloor n/2 \rfloor) \leq 1 + (\lfloor \log_2 n/2 \rfloor + 1) \leq 1 + \lfloor \log_2 n \rfloor.$$

Démonstration 4.3.0.2. —
Par récurrence sur $k = \lfloor \log_2 n \rfloor$:

— si $k = 0$ alors $n = 1$ or on sait que $C(1) = 1$ donc c'est vérifié ;
— si $k = 1$ alors $n = 2$ et on a bien $C(2) = 2$;
— supposons $k \geq 2$ et la propriété vraie pour tout n tel que $k \geq \lfloor \log_2 n \rfloor$;
— soit n tel que $k < \lfloor \log_2 n \rfloor \leq k + 1$,

$$\begin{aligned}
C(n) &\leq 1 + C(\lfloor n/2 \rfloor) \\
C(n) &\leq 1 + \lfloor \log_2 \lfloor n/2 \rfloor + 1 \rfloor \\
C(n) &\leq 1 + \lfloor \log_2 \lfloor n/2 \rfloor + \log_2 \lfloor n/2 \rfloor \rfloor \\
C(n) &\leq \lfloor 1 + \log_2(2 \times \lfloor n/2 \rfloor) \rfloor \\
C(n) &\leq \lfloor 1 + \log_2(n) \rfloor
\end{aligned}$$

4.4. Preuve de correction

Il faut vérifier que :

1. l'algorithme termine toujours (i.e. la taille des tableaux diminue strictement) ;
2. l'algorithme renvoie bien la bonne position si, et seulement si, x est dans le tableau.

DÉMONSTRATION 4.4.0.3. —
Soit T un tableau, l la borne inférieure et u la borne supérieure. On fait une preuve par récurrence sur $u - l$.

- Si $u < l$ alors l'algorithme est correct, puisqu'il renvoie *none*.
- Sinon, $l \leq m \leq u$ avec $m = u$ si, et seulement si, $l \leq u + 1$, c'est-à-dire $l = u$ ou $l = u + 1$. Donc $u - l > (m - 1) - l$ et $u - l > u - (m + 1)$ ce qui donne les tailles d'intervalles des appels récursifs. Ainsi les appels récursifs sont effectués sur des intervalles strictement plus petits, donc l'algorithme s'arrête.

DÉMONSTRATION 4.4.0.4. —
On le fait aussi par récurrence sur la taille du tableau trié T, $n = u - l + 1$, sur lequel porte la recherche dichotomique.

- Si $u - l + 1 \leq 0$ alors l'algorithme renvoie *none* et c'est la bonne valeur.
- Si $u - l + 1 = 1$, c'est-à-dire $u = l$ alors l'algorithme renvoie m si $T[m] = x$ et *none* dans le cas contraire ce qui est dans les deux cas la bonne réponse.
- Hypothèse de récurrence : on suppose que l'algorithme renvoie la bonne réponse pour $n = u - l + 1 \geq 1$. On doit montrer la propriété pour $n + 1$.
  - Puisque $n + 1 > 1$ on a $l \leq m < u$.
  - Si la condition est vérifiée, l'algorithme renvoie bien n.
  - Sinon un appel récursif est effectué avec un sous-tableau de T de taille strictement inférieure à $n + 1$ et par hypothèse de récurrence on conclut.