Introduction à l'algorithmique et la complexité (et un peu de CAML)

Algorithmes de Tri (et leur complexité)

Nicolas Nisse

Université Côte d'Azur, Inria, CNRS, I3S, France

Cours dispensés en MPSI (option Info) au CIV, depuis 2011-

http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/

Outline

Tri par **selection** des éléments d'un tableau

Étant donné un tableau $T = [t_0, \cdots, t_{n-1}]$ d'éléments d'un ensemble ordonné (disons d'entiers), on veut modifier la position des éléments dans T tel, qu'à la fin, les éléments de T soient ordonnés (e.g., du plus petit au plus grand) (tri sur place).

Algorithme par Sélection: cf. Slide 12 Chap. 2 (fonction "Echange" : slide 17, Chap. 1).

```
#let triSelection t =
        let n = vect_length t in
        for i = 0 to n-2 do
                let min_courant = ref t.(i) in
                let index_courant = ref i in
                for j=(i+1) to (n-1) do
                        if t.(j)< !min_courant then
                                begin
                                min_courant := t.(j);
                                index_courant := j;
                                end
                done;
                Echange t i !index_courant;
        done;
        t;;
triSelection : 'a vect -> 'a vect = <fun>
```

Complexité : Double boucles imbriquées.
Notons que $c(Echange) = O(1)$. Donc,
$c(triSelection) =$
$$O(1) + \sum_{i=0}^{n-2}(O(1) + \sum_{j=i+1}^{n-1} O(1)) = O(n^2).$$

Déterminer la complexité d'un algorithme consiste à compter le nombre "d'opérations élémentaires" réalisées. Ici, on peut se dire que ce qui compte c'est le nombre de "déplacements" des éléments dans le tableau (c-à-d, le nombre d'exécutions de $Echange(t, i, !index\_courant)$ lorsque $i \neq !index\_courant$).
Si une "opération élémentaire" est définie comme un tel déplacement :

- Quelle est la complexité de l'algorithme *triSelection* appliqué au tableau $[|1, 2, 3, \cdots, n|]$?
- Montrez que le pire cas a lieu pour le tableau $[|n, n-1, n-2, \cdots, 1|]$

Souvent (dans ce cours), ce qui importe pour le tri est le nombre de comparaisons effectuées.

Outline

Tri par **insertion** des éléments d'un tableau

Algorithme par Insertion : Initialement, vous tenez toutes vos cartes (non triées) dans votre main droite. À chaque étape, vous prenez la "première" (la plus à gauche) carte qui se trouve dans votre main droite, et l'**insérez** "à sa place" dans votre main gauche. À la fin, vos cartes sont triées par ordre croissant dans votre main gauche !!

```
#let triInsertion t =
        let n = vect_length t in
        for i = 1 to n-1 do
                let j = ref (i-1) in
                while (!j>=0) && (t.(!j)>t.(!j+1)) do
                        Echange t !j (!j+1);
                        j := !j -1;
                done;
        done;
        t;;
triInsertion : 'a vect -> 'a vect = <fun>
#let t = [| 7;8;2;5;3;6|];;
t : int vect = [|7; 8; 2; 5; 3; 6|]
#triInsertion t;;
- : int vect = [|2; 3; 5; 6; 7; 8|]
```

Rappel: fonction "Echange" (slide 17, Chap 1).

Terminaison : preuve laissée au lecteur.

Correction : Un invariant de boucle est "après l'itération i, le sous-tableau $[|t.(0), \cdots, t.(i)|]$ est ordonné"
À prouver par récurrence sur i.

Complexité : Le nombre d'applications de "Echange" est

$$c(triInsertion) = O(\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} O(1)) = O(n^2).$$

Un pire cas est $t = [|n, n-1, n-2, \cdots, 1|]$.

Outline

Tri à bulles

Algorithme de tri à bulles : comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. (voir animation sur Wikipedia). Intuitivement, on fait "remonter" (comme des bulles) les éléments les plus grands jusqu'à ce qu'ils atteignent leur place.

Rappel: fonction "Echange" (slide 17, Chap 1).

```
#let triBulles t =
        let n = vect_length t in
        for i = (n-1) downto 1 do
                for j=0 to (i-1) do
                        if t.(j+1)<t.(j) then Echange t j (j+1);
                done;
        done;
        t;;
triBulles : 'a vect -> 'a vect = <fun>
```

Terminaison : preuve laissée au lecteur.

Correction : Un invariant de boucle est "après l'itération i, le sous-tableau $[|t.(i), \cdots, t.(n-1)|]$ contient les $n-i$ plus grands éléments de t qui sont ordonnés".
À prouver par récurrence (descendante) sur i

Complexité : Le nombre d'applications de "Echange" est $O(n^2)$.
Un pire cas est $t = [|n, n-1, n-2, \cdots, 1|]$.

Outline

Tri Fusion (*"merge sort"* en anglais)

Pour changer, nous trions ici une **liste** et les algorithmes sont présentés sous forme récusive.

Algorithme de tri fusion : Divisons notre liste $\ell = [u_0, \cdots, u_{n-1}]$ en deux. Trions (récursivement) les listes $\ell_1 = [u_0, \cdots, u_{(n-1)/2}]$ et $\ell_2 = [u_{(n-1)/2+1}, \cdots, u_{n-1}]$. Fusionnons les listes triées ℓ_1 et ℓ_2 de façon à obtenir une nouvelle liste ℓ' contenant les éléments de ℓ triés.

```
#let rec divise l = match l with
        |[] -> ([],[])
        |[e]-> ([e],[])
        |a::b::r -> let (l1,l2) = divise r in
                    (a::l1,b::l2);;
divise : 'a list -> 'a list * 'a list = <fun>
#let rec fusion l1 l2 = match (l1,l2) with
        |l,[] -> l
        |[],l -> l
        |a::r1,b::r2 -> if a<b then a::(fusion r1 l2)
                        else b::(fusion l1 r2);;
fusion : 'a list -> 'a list -> 'a list = <fun>
#let rec tri_fusion l = match l with
        |[] -> []
        |[e]-> [e]
        | l -> let (l1,l2)=divise l in
               fusion (tri_fusion l1) (tri_fusion l2);;
tri_fusion : 'a list -> 'a list = <fun>
```

"divise" prend une liste ℓ de longueur n et crée 2 nouvelles listes de longueur $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$ contenant les éléments de ℓ.

"fusion" prend 2 listes **triées** et rassemble leurs éléments dans une nouvelle liste triée.

"tri_fusion" prend une liste et, en utilisant les fonctions précédentes, crée une nouvelle liste contenant les éléments de ℓ triés.

Terminaison : preuve par récurrence laissée au lecteur.

Correction : preuve par récurrence laissée au lecteur.

Exercices : transposez ces algorithmes : sous forme itérative, pour trier des tableaux.

Complexité du Tri Fusion

Nous nous intéressons principalement au nombre de comparaisons des éléments de la liste (i.e, "opération élémentaire" = comparaison).

Divise : Soit $c_d(n)$ la complexité de "divise" appliquée à une liste de longueur n. Alors, $c_d(0) = X$ et $c_d(n) = X + c_d(n-2)$ avec $X = 0$ si on ne compte que les comparaisons et $X = O(1)$ si on compte aussi la création/l'ajout d'un élément dans une liste. Donc, par récurrence, $c_d(n) = 0$ ou $c_d(n) = n/2 = O(n)$ selon ce que l'on compte (le choix de ce que l'on compte n'aura pas d'influence sur la suite).

Fusion : Soit $c_f(n_1, n_2)$ la complexité de "fusion" appliquée à 2 listes de longueur n_1 et n_2. Alors, $c_f(0,0) = c_f(n_1,0) = c_f(0,n_2) = O(1)$ et $c_f(n_1,n_2) \leq O(1) + \max\{c_f(n_1-1,n_2); c_f(n_1,n_2-1)\}$. On en déduit, par récurrence, $c_f(n_1,n_2) = O(n_1+n_2)$.

Tri_fusion : Soit $c(n)$ la complexité de "tri_fusion" appliquée à une liste de longueur n. Alors $c(0) = c(1) = O(1)$ et $c(n) = c_d(n) + c(\lceil n/2 \rceil) + c(\lfloor n/2 \rfloor) + c_f(\lceil n/2 \rceil, \lfloor n/2 \rfloor)$ (application de "divise" à ℓ, puis 2 applications de "tri_fusion" aux listes ℓ_1 et ℓ_2 résultantes, de longueur $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$ respectivement, et enfin "fusion" des 2 listes triées obtenues). La façon de résoudre une telle suite est présentée ci-dessous.

Posons $u_k = c(2^k)$. On a $u_0 = O(1)$ et $u_k = 2u_{k-1} + O(2^k)$. Soit $v_k = \frac{u_k}{2^k}$. Alors, $v_0 = O(1)$ et $v_k = v_{k-1} + O(1)$. Donc, par récurrence, $v_k = O(k)$. On en déduit $u_k = O(k2^k)$. Pour conclure, pour tout $n \in \mathbb{N}$, soit $k \in \mathbb{N}$ tel que $2^{k-1} < n \leq 2^k$. Alors, prouvez que $c(n) \leq u_k = O(k2^k)$. Ainsi, $c(n) = O(k2^k) = O(n \log n)$.

L'algorithme de Tri_fusion réalise $O(n \log n)$ comparaisons pour trier une liste de longueur n.

Peut on faire mieux ?

Outline

Faire mieux que le Tri Fusion ?

Soit *tab* un tableau de n entiers à trier. Supposons que les éléments de *tab* sont $\leq k \in \mathbb{N}$.

```
#let triAvecBorneSup tab k =
        let n = vect_length tab in
        let tmp = make_vect (k+1) 0 in
        for i=0 to n-1 do
                tmp.(tab.(i)) <- tmp.(tab.(i)) +1;
        done;
        let res = make_vect n 0 in
        let current = ref 0 in
        for j=0 to k do
                if tmp.(j)>0 then
                        for i=1 to tmp.(j) do
                                res.(!current) <- j;
                                current := !current +1;
                        done;
        done;
        res;;
triAvecBorneSup : int vect -> int -> int vect = <fun>
```

Les peuves de terminaison et de correction de cet algorithme sont laissées au lecteur.

Sa complexité est $O(n+k)$. Il a donc une meilleure complexité que Tri_fusion si $k = o(n \log n)$, mais il demande la connaissance d'une borne supérieure sur les éléments de t.

De plus, en espace, il crée un tableau de taille k qui, si $n = o(k)$, est plus coûteux que l'espace nécessaire à Tri_fusion.

Sans information sur les éléments de *tab*, l'algorithme de Tri_fusion est optimal dans le pire cas.

Intuition de Preuve : On suppose que $tab = [|t_1, \cdots, t_n|]$ contient des éléments distincts deux-à-deux.
Trier *tab* revient à trouver la permutation $\pi : [1,n] \to [1,n]$ telle que le tableau $tab' = [|t_{\pi(1)}, t_{\pi(2)}, \cdots, t_{\pi(n)}|]$ est trié, i.e., pour tout $1 \leq i < j \leq n$, $t_{\pi(i)} < t_{\pi(j)}$.

Le nombre de permutations $\pi : [1,n] \to [1,n]$ est $n! \approx n^n$.
Soient $1 \leq i < j \leq n$ et soient Π_1 l'ensemble des permutations telles que $t_{\pi(i)} < t_{\pi(j)}$ et Π_2 l'ensemble des permutations telles que $t_{\pi(i)} > t_{\pi(j)}$. Il y a une bijection (laissée au lecteur) de Π_1 vers Π_2. Donc $|\Pi_1| = |\Pi_2| = (n-1)!/2$.
Autrement dit, effectuer une comparaison permet donc de diviser le nombre de permutations candidates par 2.

Trier notre tableau revient à faire des comparaisons jusqu'à ne laisser qu'une unique permutation candidate. Soit x le nombre minimum de comparaisons, il doit satisfaire : $n!/2^x \leq 1$. Donc, à la louche, $n^n \leq 2^x$. En prenant le logarithme, on a donc

$$n \log n \leq x = \#\textit{min de comparaisons}.$$