IUT De Villetaneuse
Dépt informatique

Cours d'algorithmique
2éme Année

Cours 7

Mesures de performance – exemple des tris

Les tris

On a un ensemble de données que l'on veut trier. Les données sont dans un tableau de taille n. En pratique, n est de l'ordre des millier, ou même des millions. La plupart des algorithmes s'astreignent donc à ne pas demander d'espace supplémentaire et à procéder par **permutation** (échange d'éléments) dans le tableau.

Deux tris naïfs

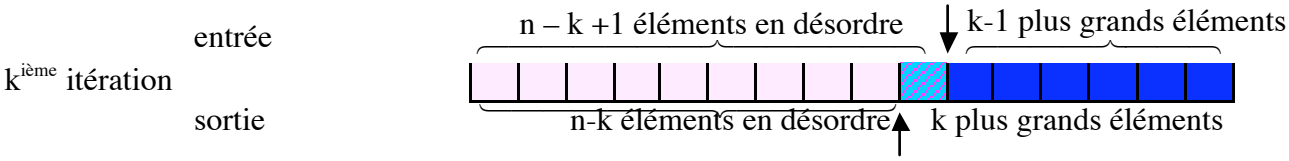
Exemple1 : tri par recherche du maximum

Le principe de l'algorithme est le suivant : on cherche dans le tableau le plus grand élément ; on le met en place (il doit devenir le dernier) en le permutant avec le dernier ; puis on recommence avec le tableau sauf le dernier (le terme technique pour recommencer est : **itérer**).

L'évolution de l'algorithme est plus facile à comprendre si l'on regarde l'état du tableau à la fin de la kieme itération. : les k plus grands éléments sont déjà à leur place définitive.

Au début du programme :

Tous les éléments en désordre dans le tableau



La kieme itération se contente de choisir le plus grand élément dans la partie non triée, et de le mettre en place – donc la taille de la partie triée augmente de 1. Au bout de n itérations, le tableau est trié.

Le temps consommé peut être estimé :
- temps pour rechercher le maximum dans un tableau de taille t :
- temps pour permuter 2 éléments :
- temps pour les n itérations :

Donc on a une complexité en $O(n^2)$: si on trie 10 fois plus d'éléments, le temps de calcul est multiplié par environ …
Si l'on trie 500 éléments en 1s, il faudra pour en trier 50 000 environ

Exemple 2 : tri à bulles

Le principe est le suivant : on compare tour à tour chaque élément avec son voisin, et on les échange si ils ne sont pas dans l'ordre. On s'arrête quand il n'y a plus d'échange possible. Cette première idée n'est en fait pas assez précise pour définir un algorithme : après une comparaison (et peut-être un échange) on n'a pas dit quel doit être la suivante. Dans le tri à bulle, on parcourt tout le tableau du début à la fin. Puis on revient au début pour une nouvelle itération, On s'arrête quand une itération n'a donné lieu à aucun échange.

Le tri à bulles n'est pas très efficace. Nous ne prendrons pas le temps d'analyser sa complexité, mais elle est en $O(n^2)$. Pour en avoir une idée, on peut regarder ce qui se passe quand le tableau de départ est trié à l'envers.

Deux améliorations importantes

Exemple 3 : tri par tas

C'est la même idée que le tri par recherche du maximum : à chaque itération, on trouve le plus grand et on le met à sa place. Mais on réutilise l'idée qui nous a servi dans les files de priorité : si on organise bien les données en arbre, on trouve le plus prioritaire en Log n au lieu de n, y compris la réorganisation de l'arbre – il suffit de dire que le plus prioritaire est le plus grand. Prélever les n éléments du tableau l'un après l'autre est de complexité O(n Log n).

Il faut aussi compter le temps pour organiser les données, ce qu'on peut faire en créant une file de priorité vide puis en y insérant les données l'une après l'autre. Comme chaque insertion est de complexité O(Log n) (moins en fait, puisque le nombre de nœuds de l'arbre dans lequel on insère n'est pas n), l'ensemble est encore en O(n Log n).

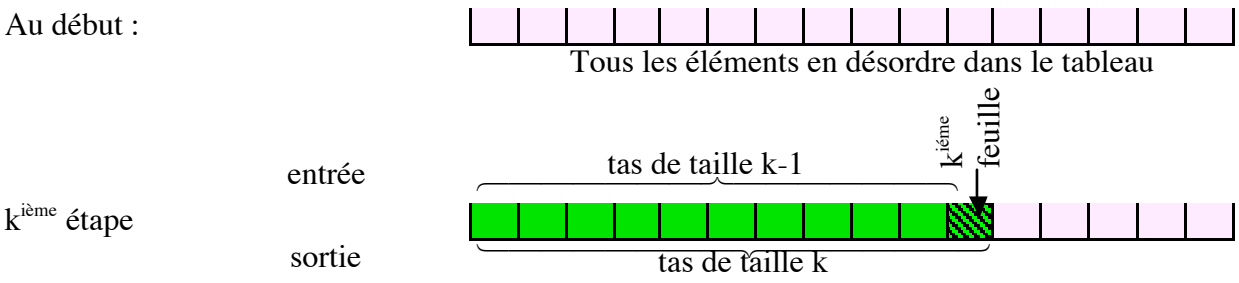
Comme les données sont dans un tableau, on réutilisera facilement l'autre idée du cours 5 : implémenter la file de priorité dans un tableau (donc la file de priorité sera un tas).

Au départ les données sont comme toujours en désordre dans le tableau. On a 2 phases :

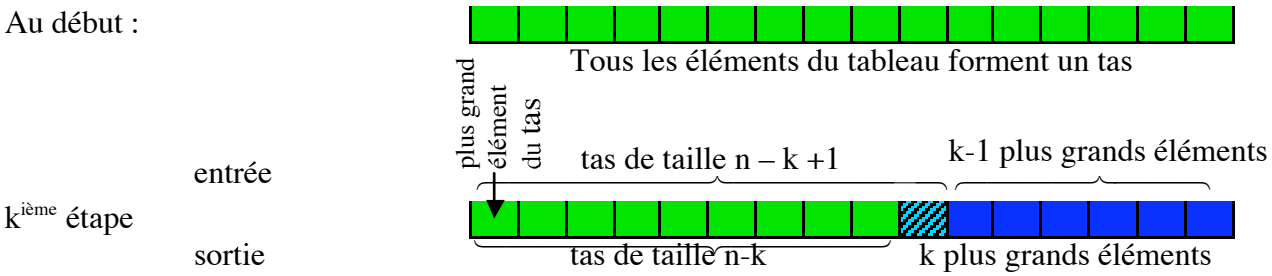
i) construire le tas en insérant les nœuds un par un

ii) tri par recherche du maximum dans un tas : prélever le plus grand élément et le mettre à la fin, à la place de la feuille qui sert à reconstruire le tas.

Phase i :



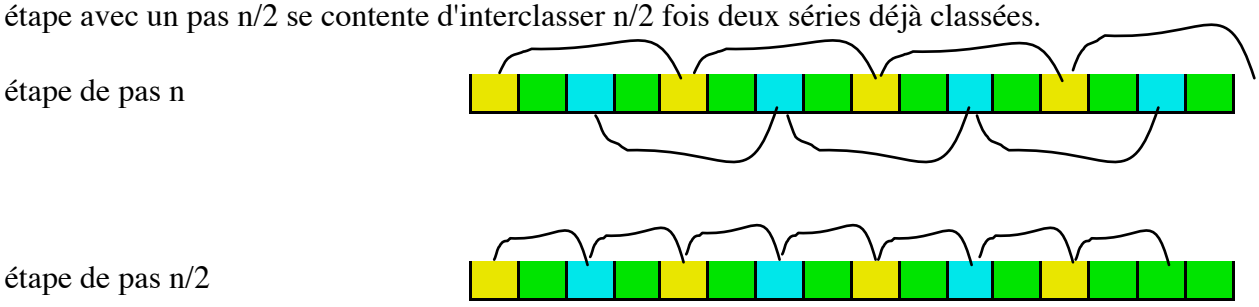
Phase ii :



Quand on a implémenté correctement une file de priorité dans un tableau, écrire un tri à bulle ne demande que quelques lignes de code. On a pourtant beaucoup gagné : si l'on multiplie le nombre de données par 100, on multiplie le temps par moins de $100.\log_2(100) \cong 600$ (au lieu de 10000 pour le tri par recherche du maximum).

Exemple 4 : Tri à bulles à pas variable

A chaque échange, le nombre fait un pas vers sa place définitive (ou parfois en sens contraire). Dans le tri à bulles, tous les pas sont de une case. Pour améliorer cela, le tri à bulles à pas variable utilise un principe dichotomique : au début, les pas sont les plus longs possibles. Puis quand une itération ne change plus rien, on divise le pas par deux. On termine avec un pas de 1, mais on a été plus vite.
Des optimisations sont possibles : si l'on choisit comme pas initial une puissance de 2, chaque étape avec un pas n/2 se contente d'interclasser n/2 fois deux séries déjà classées.



Donc chaque itération range correctement au moins un élément à chaque extrémité de la série, soit n/2 éléments à chaque extrémité pour une itération de pas n/2. On peut réduire le parcours de l'itération de n à chaque fois, et cela limite aussi le nombre d'itérations

Mesurer les performances

Les calculs de complexité donnent une idée des performances moyennes d'un algorithme. Une technique complémentaire consiste à mesurer les performances d'un programme, dans la réalité, pendant qu'il tourne. C'est un moyen très important de mise au point.

Trouver un jeu de tests

C'est la première tâche. On ne mesure jamais sur un seul exemple, parce que le résultat peut être du au hasard et n'avoir que très peu de signification. Chaque exemple s'appelle un test, un ensemble d'exemples utilisés ensemble s'appelle un **jeu de tests**. Il faut choisir des jeux de test **représentatifs de la réalité** pour que les mesures soient valables. C'est un point délicat (qui se pose aussi dans les tests d'absence de bug). On a deux grandes approches.

Tests sur données réelles

On dispose parfois (par exemple dans les archives d'une entreprise) de données tout à fait analogues à celles que l'on aura à traiter. On peut essayer le programme sur ces données anciennes pour voir si on obtient un résultat correct dans un temps acceptable.

Jeux de tests aléatoires

L'autre solution consiste à fabriquer un ou plusieurs jeux de tests en tirant au sort la valeur des données. Une série de nombres pseudo aléatoire est une série telle qu'un humain ou une machine qui ne connaît pas la fonction qui les engendre peut difficilement prévoir la suite. On dispose maintenant de fonctions bien faites pour générer des séries de nombres pseudo aléatoires. En Java, la fonction Math.random() (renvoie un double entre 0 et 1). La classe Random a des méthodes nextInt(), nextFloat(), etc. qui fournissent des séries pseudo-aléatoires de tous les types primitifs sauf char et String.
Vérifier que les jeux de test aléatoires que l'on va générer sont représentatifs demande une étude précise. Nous nous contenterons pour l'instant de méthodes approchées.

Mesurer le temps utilisé par le programme

La façon la plus simple de connaître le temps mis par le programme pour s'exécuter, c'est de regarder l'heure au début et à la fin ! En Java, la classe System dispose d'une méthode **long currentTimeMillis()** qui donne l'heure au millième de seconde près.
C'est une méthode simple, mais rudimentaire : presque toutes nos machines sont multitâches et multiutilisateur. Le temps mesuré a en fait été partagé avec d'autres processus dont on ne sait rien, et on peut obtenir un temps élevé parce qu'un autre utilisateur a lancé un gros calcul ! De plus, on ne fait pas la distinction entre le temps de calcul et le temps d'entrée/sortie.
En C, on mesure le temps en positionnant un timer qui envoie un signal à intervalle régulier. On dispose de trois timers, selon que l'on veut mesurer le temps réel, le temps de calcul ou le temps de (calcul + Entrées/Sorties).

Compter le nombre d'opérations

Il faut au départ écrire le code en exécutant les opérations importantes dans des fonctions. Par exemple, au lieu de coder pour un tri (on suppose pour l'exemple qu'on trie des entiers qui sont dans tabint[]) :

```
if (tabint[i] < tabint[j]) {
        int tmp = tabint[i] ;
        tabint[i] = tabint[j] ;
        tabint[j] = tmp ;
}
```

code qu'on risque de retrouver plusieurs fois dans l'algorithme, il vaut mieux écrire :

```
boolean before(int i, int j) {
        return tabint[i] < tabint[j] ;
}

void swap(int i, int j) {
        int tmp = tabint[i] ;
        tabint[i] = tabint[j] ;
        tabint[j] = tmp ;
}

if (before(i,j)
        swap(i,j) ;
```

C'est plus lisible, on ne réécrira pas plusieurs fois les comparaisons et les affectations, donc on a moins de chance de faire une erreur. De plus, on peut très facilement étudier le comportement du programme en comptant le nombre de comparaisons et de permutations (en général, le nombre d'opérations de chaque sorte).

En Java :

Dans la classe qui implante la méthode de tri, ajouter deux variables membres *nbCompare* et *nbPermute* de type *protected int* et initialisées à 0.
Modifier le code de before() et de swap() :

```
boolean before(int i, int j) {
        nbCompare++ ;
        return tabint[i] < tabint[j] ;
}
```

```
void swap(int i, int j) {
        nbPermute++ ;
        int tmp = tabint[i] ;
        tabint[i] = tabint[j] ;
        tabint[j] = tmp ;
}
```

à la fin du tri, afficher nbCompare et nbPermute

En C

On utilise des variables statiques de portée locale (elles sont initialisées à 0 et ont une durée de vie permanente, mais ne sont accessibles que dans la fonction) ;

```
boolean before(int i, int j) {
        static int nbCompare ;
        nbCompare++ ;
        return tabint[i] < tabint[j] ;
}
```

```
void swap(int i, int j) {
        static int nbPermute ;
        nbPermute++ ;
        int tmp = tabint[i] ;
        tabint[i] = tabint[j] ;
        tabint[j] = tmp ;
}
```

Ce code n'est pas tout à fait complet : il faut prévoir un mécanisme pour récupérer les résultats, par exemple un appel de fonction avec un argument spécial [ici, (–1,-1) convient bien].

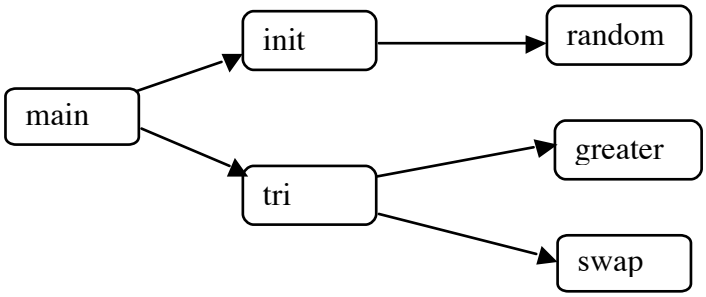
Profiler un programme

Profiler, c'est utiliser un outil système pour suivre l'exécution du programme et faire des statistiques sur le temps passé dans chaque fonction ou partie de programme.
En C, on compile avec l'option –pg, par exemple **gcc –pg headtail.c**. Quand on exécute **a.out**, l'exécution engendre un fichier de trace **gmon.out**. On peut alors voir des statistiques de comportement des fonctions avec **gprof a.out** . le tableau qui suit a été obtenu en lançant un tri à bulles sur un tableau de 10000 entiers.

(f) veut dire : fonction seule ; (f&d) abrège : fonction et descendants.

% temps (f)	secondes cumulées (f & d)	secondes seul (f)	appels	nbre moyen de ms par appel (f)	nbre moyen de ms par appel (f&d)	nom [index]
14,8	10.13	10.13	99150084	0.00	0.00	_greater[4]
5,6	32.63	3.85	1	3850.00	32630.00	_tri[3]
27,3	18.57	18.57	24784824	0.00	0.00	_swap[5]
0.0	35.38	0.01	1	10.00	35380.00	_init[6]
0.0	68.12	0.00	1	0.00	35380.00	_main[1]

On peut ainsi mesurer quelles sont les parties du programme le plus souvent utilisées et celles qui consomment le plus de temps, donc celles qu'il est le plus important d'améliorer.
Dans l'exemple, l'arborescence des appels est :



Tout le temps consommé par init() l'est en fait par la fonction de bibliothèque random() qui n'est pas tracée. swap() consomme 2 fois plus de temps que greater() qui pourtant est appelé 4 fois plus. À eux deux ils représentent 90% du temps de tri(). A moins d'améliorer leur implémentation (en codant par exemple en assembleur), on ne peut pas gagner grand chose ici sur le codage. Mais le profilage ne montre pas qu'on peut améliorer l'algorithme !

En java :
le JDK comporte une API de profilage – mais pas d'utilitaire pour profiler.