  
Cahier de TD de SDD: Méthodes de tri

TRIS

Exercices 4.1 : tris par tas simples 🟅🟅

Ordonner les tableaux suivants à l’aide du tri par tas :

A – {7, 3, 1, 2, 3}

B – {13, 1, 3, 31, 11}

C – {6, 4, 3, 2, 1}

D – {2, 3, 4, 2, 3}

E – {5, 4, 3, 2, 1}

Exercices 4.2 : tri par tas plus compliqués (mais même principe) 🟅🟅

Ordonner les tableaux suivants à l’aide du tri par tas :

A – {44, 55, 12, 42, 94, 18, 6, 67} (Exemple du cours)

B – {29, 23, 17, 11, 5, 2, 3, 7, 13, 19, 25, 31} (CE 2 SDD L2 2012)

Exercice 4.3 : tri rapide 🟅🟅

Ordonner le tableau suivant à l’aide du tri rapide :

{8, 5, 9, 4, 2, 3, 7, 10, 6, 1} (Exemple du cours)

Exercices supplémentaires pour les plus courageux

Exercice 4.4 : travail de recherche personnel : complexité comparée des tris 🟅🟅

Comparer la complexité algorithmique des tris les plus connus.

Quel sont leurs avantages et inconvénients respectifs ?

Sujet TP 4.5 : performance comparée de tris 🟅🟅🟅

Ce sujet, plus simple que les deux précédents, complète la dernière séance de TD.

L’objectif est de programmer plusieurs méthodes de tri et de comparer leurs performances respectives.

Votre programme doit permettre de générer pour quelques tris, un rapport de performances comparées.

Facile à charger sous Excel, ce rapport permet la génération de graphiques.

Il est indispensable de réaliser au moins le tri pas tas, et soit le tri par insertion, soit le tri rapide.

Votre travail sera apprécié davantage sur la qualité de votre exécutable et sur celle de vos rapports Excel (que sur la reproduction des algorithmes standards disponibles sur Internet).

A vous d’élaborer la meilleure façon de « faire parler » graphiquement vos données.

Principe de fonctionnement

Votre programme fonctionne en deux phases ou se présente sous la forme de deux exécutables.

Les deux phases produisent chacune un fichier de données destiné à être chargé sous Excel.

Chaque phase démarre par la saisie par l’utilisateur d’un paramètre *n* entier :

Phase 1 – Test qualitatif de réaction des méthodes de tri à l’ordre initial du tableau :

* Le test porte sur toutes les permutations du tableau { 1, …, n }.

Phase 2 – Test de montée en charge :

* Le test porte sur des tableaux de taille croissante de 1 à *n*.
* A chaque itération, il s’agit de réaliser les trois tests comparatifs suivants :
  + tableau pré-ordonné croissant { 1, …, n },
  + tableau pré-ordonné décroissant { n, …, 1 },
  + tableau à valeurs entières aléatoires.

Données à produire

Dans chacun des deux fichiers, chaque ligne représente un test comparatif.

Les informations suivantes sont enregistrées sur chaque ligne du premier fichier :

* Un numéro de ligne,
* Le tableau initial avant le tri,
* [Eventuellement la permutation donnée sous forme de produit de cycles à supports disjoints],
* Pour chaque tri sur ce même tableau de départ :
  + Le nombre de transpositions effectuées,
  + [Eventuellement la liste de ces transpositions (un plus qui sera apprécié)],

Voici un exemple :

HEAP SORT QUICK SORT

ID ARRAY PERMUT #T TLIST #T TLIST

3.1 { 0 1 2 } (0)(1)(2) 4 (0 2)(0 2)(0 1)(0 1) 3 (1 1)(0 0)(1 1)

3.2 { 0 2 1 } (0)(1 2) 3 (0 1)(0 2)(0 1) 3 (1 1)(0 0)(1 2)

3.3 { 1 0 2 } (0 1)(2) 3 (0 2)(0 2)(0 1) 2 (1 1)(0 1)

3.4 { 1 2 0 } (0 1 2) 4 (0 1)(0 2)(0 1)(0 1) 3 (1 2)(1 1)(0 1)

3.5 { 2 1 0 } (0 2)(1) 3 (0 2)(0 1)(0 1) 3 (2 2)(0 2)(0 0)

3.6 { 2 0 1 } (0 2 1) 2 (0 2)(0 1) 3 (2 2)(0 2)(0 1)

Dans le second fichier, les lignes viennent par trois, correspondant aux trois cas vus plus haut.

Les informations suivantes sont enregistrées dans le second fichier :

* Un numéro de ligne de la forme <taille\_tableau>.<cas\_de\_test> (cf. les 3 cas),
* Pour chaque tri  (pensez à le dupliquer) :
  + Le nombre de transpositions effectuées,
  + Le temps d’exécution du tri en millisecondes.

Options

Les options sont facultatives, mais permettent une majoration de la note.

*Option 1 – Ajout « tri par tas en mode dégradé »*

Cf. une question récurrente posée en TD, la première option consiste à prévoir la désactivation, dans votre tri par tas, de la fonctionnalité de rétablissement, par descente récursive, de la structure de tas, lorsque celle-ci est rompue par une permutation de la boucle de second niveau de l’algorithme.

*Option 2 – Intégrer d’autres tris, choix libre dans votre benchmark.*

Problème 4.6 – Réflexion algorithmique 🟅🟅🟅🟅🟅

***Note*** *– Pour cet exercice, vous serez évalué(e) sur la qualité scientifique de votre analyse formelle. Si vous ne maîtrisez pas parfaitement le tri par tas standard, il est inutile de vous y essayer. Argumentez de la manière la plus ‘compacte’ possible en évitant les sophismes, ou n’écrivez rien et restez concentré(e) sur les autres exercices.*

Le tri par tas est basé sur la mise en correspondance d’un tableau et d’un arbre binaire.

Supposons que nous souhaitions généraliser ce tri à l’aide d’un paramètre n > 1 donné qui permet la mise en correspondance d’un tableau et d’un arbre n-aire, quelles modifications proposeriez-vous dans les étapes clés de l’algorithme (adressage des parents vers les enfants, des enfants vers les parents, fonctionnement de l’algorithme) ?

**Question subsidiaire** : pensez-vous que cette généralisation puisse permettre d’améliorer les performances de l’algorithme ? Comment ?

Rappels de cours

1 Tri par tas (Heapsort)

1.1 Définition d'un ABOV (Arbre Binaire Ordonné Verticalement)

Un arbre binaire est ordonné verticalement si pour tout nœud l'ensemble de ses descendants lui sont inférieurs.

Remarque

Un ABOV ne présente pas une relation d'ordre totale, c.-à-d. il n'existe aucun moyen déterminé à obtenir les informations dans l'ordre.

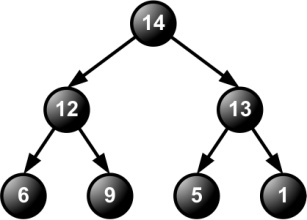


Figure 1

Pas de relation d'ordre entre deux branches.

Relation d'ordre partielle.

- Pas de possibilité systématique de

- Placer un élément

- Rechercher un élément

🡪 Moins utile.

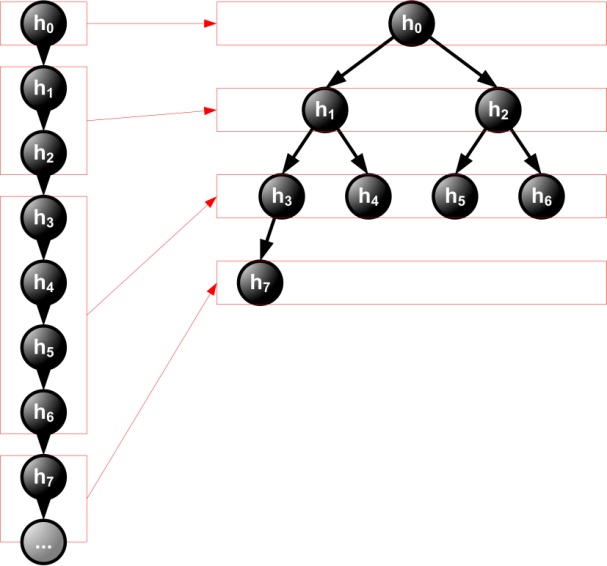
Cependant il existe une méthode de tri efficace : le tri par tas (Heap Sort).

1.2 Tris par tas

La méthode du tri par tas consiste à considérer le tableau à trier comme la représentation d'un arbre binaire stocké de manière compacte, à ordonner l'arbre verticalement, puis à trier l'arbre, c.-à-d. à extraire les extrema successifs.

Le tri par tas manipule un tableau, c.-à-d. que les données à trier sont stockées dans un tableau. On n'utilise pas de pointeurs, mais la représentation arborescente en tableau.

On crée l'arbre compact : le remplissage se fait du haut vers le bas et pour chaque étage de la gauche vers la droite.



Tas 2

On peut voir que si l'indice j ≥ N / 2 alors t[j] est forcément une feuille.

Exemples :

– Si N = 8 alors N / 2 = 4 d'où T[4], T[5], T[6] et T[7] sont des feuilles.

– Si N = 7 alors N / 2 = 3 d'où T[3], T[4], T[5] et T[6] sont des feuilles.

1.3 Principe de l'algorithme (Construction de tas)

1) Ordonner verticalement l'arbre en commençant par le 'bas' (par les plus grands indices) : On ne commence pas par les derniers car ce sont des feuilles, ils n'ont pas de fils, ce sont donc des ABOV singletons.

🡪 début à N/2(-1) /\* j 🡨 N/2 \*/

a) Pour l'élément d'indice j

1. Regarder s'il a deux fils :

Si oui, déterminer le plus grand des 2

Sinon, ne rien faire

2. SI T[j] < l'élément déterminé dans a)1 les permuter.

b) SI l'élément descendu n'est pas une feuille

Recommencer en a)1.

c) SI l'élément n'a pas été descendu ou SI l'élément descendu est une feuille

Fin pour cet élément

Retour en a) avec j-1

d) SI j=0(-1) 🡪 on a un ABOV

FIN du 1) 🡪 max en T[1(0)]

2) Echanger T[1(0)] et T[n(n-1)]

🡪 le max est dans T[n(n-1)]

3) Refaire un ABOV en considérant seulement n-1 élément et en faisant 1) seulement pour l'indice 1(0).

4) Retour en 1) si au moins 2 éléments dans le tableau

Sinon fin : le tableau est trié.

Remarque

Il s’agit d’un tri stable : Il ne dégénère pas en fonction de l'ordre des éléments à trier.

Algorithme du HEAPSORT

**Donnée** : n de type entier, i de type entier

**Donnée modifiée** : tableau d’éléments de type T

**En-tête en C** : void descendre(int i, int n, T t[]) */\* descendre un élément d'indice i dans un tableau t de n éléments \*/*

**VAR** planque : T, descente : booléen

**DEBUT**

l **←** 2 \* i + 1 *{ l est l'indice de fils gauche de l'élément i }*

planque **←** t[i]

descente **←** vrai

*// t[i] n'est pas une feuille*

**TANTQUE** descente **=** vrai **ET** l **≤** n **-** 1 **FAIRE**

**SI** l **<** n **-** 1 **ALORS** *{ il y a 2 fils }*

**SI** t**[**l **+** 1**] >** t**[**l**] ALORS**

*{ l est l'indice de l'élément à comparer avec l'élément i }*

l **←** l **+** 1

**SI** t**[**l**] >** planque **ALORS**

t[i] **←** t**[**l**]**

i **←** l

l **←** 2 **\*** i **+** 1 *{ descente dans l'arbre }*

t**[**i**]** **←** planque

**SINON**

descente **←** faux

**FINSI**

**FINSI**

**FINSI**

**FAIT**

**FIN**

**Donnée** : n de type entier

**Donnée modifiée** : tableau d’éléments de type T

**En-tête en C** : void heapsort(int n, T t[n])

Version Cormen like (proposition : préférable – à discuter et trancher rapidement) [Note : il y a une erreur sur cet algo dans le Cormen (vieille version de 1990:/) cf. annotation marge – code C propre d’implém. A dispo.]

**TRIER-TAS(**A)

**DEBUT**

**POUR** j **←** n / 2 - 1 **A** 0 **PAR PAS DE** –1 **FAIRE**

descendre**(**j**,** n**,** t**)**

**FIN POUR**

**POUR** j **←** n - 1 **A** 1 **PAR PAS DE** –1 **FAIRE**

échanger**(**t**[**0**],** t**[**j**])**

descendre**(**0**,** j **-** 1**,** t**)**

echanger**(**t**[**0**],** t**[**1**])**

**FIN POUR**

**FIN**

2 Tri rapide (Quicksort)

**Principe** : La méthode du Quicksort due au C.A. Hoare (1962) utilise les principes généraux de division et équilibrage (paradigme algorithmique « diviser pour régner »).

2.1 Méthode du Quicksort

On veut trier A0, A1, A2, …, An-1

On suit la procédure suivante :

On choisit un élément particulier appelé pivot

On réorganise le tableau de telle sorte que

le pivot soit en As

Ai ≤ As si i < s

Ai > As si i > s

On trie A0, A1, A2, …, As-1 (par la même méthode)

On trie As+1, As+2, …, An-1 (par la même méthode)

Une première version du tri rapide est alors :

**D** : i, j : entier

**D.M.** : A : tableau[n] de T

**R** : -

**En-tête en C** : void tri\_rapide (int i, int j, T A)

**SI** j **>** i **ALORS**

partition**(**i**,** j**,** s**,** A**)**

tri\_rapide**(**i, s - 1, A**)**

tri\_rapide**(**s **+** 1**,** j**,** A)

**FINSI**

L'algorithme partition choisit un pivot et réorganise le tableau autour de ce pivot.

2.2 Problèmes du tri rapide

Il faut toujours choisir un pivot tel que les deux parties soient de tailles à peu près égales (équilibrage). Si elles sont très inégales la récursivité conduit à une grande profondeur d'appel et donc à une occupation mémoire (pile) importante.

Le tri rapide est très efficace mais peut toujours dégénérer en cas de mauvais choix du pivot.

La "machinerie" de tri rapide (récursivité, partition) est lourde et donc peu intéressante pour de petits tableaux. En partitionnant, quand on arrive à une taille inférieure à un seuil donné, il faut terminer le tri à l'aide d'une méthode d'insertion simple.

La deuxième version du tri rapide :

**D** : i, j : entier

**D.M**. : A : tableau[n] de T

**R** : -

**En-tête en C** : void tri\_rapide (int i, int j, T A)

**SI** j **-** i **>** seuil **ALORS**

partition**(**i**,** j**,** s**,** A**)**

tri\_rapide(i, s **-** 1**,** A**)**

tri\_rapide**(**s **+** 1**,** j**,** A**)**

**SINON**

tri\_insertion**(**i**,** j**,** A**)**

**FINSI**

2.3 Réalisation de partition

Partition doit choisir un bon pivot et réorganiser efficacement le tableau. Un élément du tableau est un bon pivot, s'il y a à peu près autant de valeurs inférieures que de valeurs supérieures à cet élément dans le tableau. Une solution pour déterminer un pivot à moindre coût consiste à prendre quelques éléments au hasard dans le tableau et à choisir l'élément médian.

Le pivot étant choisi, il faut organiser le tableau sans le classer. Pour réorganiser efficacement, il faut partir des deux extrémités et échanger les éléments quand ils ne sont pas classés. La réorganisation du tableau peut s'effectuer selon la méthode de Sedgewick (1975) ; elle comporte les étapes suivantes :

Placer le pivot à gauche (échanger Ai et As)

Partitionner le tableau en partant des extrémités Ai et Aj, ce qui donne un élément de séparation

Echanger l'élément de séparation obtenu avec pivot, ce qui ramène le pivot à sa place.

**D** : i, j : entier

**D.M.** : A : tableau[n] de T, int s

**R** : de type entier

**En-tête en C** : int tri\_rapide (int i, int j, T A)

s **←** choix\_pivot(A, i, j) /\* s indice initial du pivot choisi \*/

échanger A[i] et A[s]

gauche **←** i

droite **←** j

TANT QUE gauche ≤ droite

{

TANT QUE (gauche < droite) et (A[gauche] ≤ A[i])

gauche **←** gauche + 1

TANT QUE (gauche ≤ droite) et (A[droite] > A[i]) FAIRE

droite **←** droite – 1

SI gauche < droite ALORS

échanger A[gauche] et A[droite]

FINSI

échanger A[droite] et A[i]

s **←** droite /\* s indice définitif du pivot \*/

return s

Exemple

8, 5, 9, 4, 2, 3, 7, 10, 6, 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 5 | 9 | 4 | 2 | 3 | 7 | 10 | 6 | 1 |

pivot 🡨 5

s = 1

i = 0

j = 9

gauche 🡨0

droite 🡨 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 8 | 9 | 4 | 2 | 3 | 7 | 10 | 6 | 1 |

gauche 🡨 1

Échanger A[gauche] avec A[droite] (8 avec 1)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 1 | 9 | 4 | 2 | 3 | 7 | 10 | 6 | 8 |

gauche 🡨 2

droite 🡨 5

Échanger A[gauche] avec A[droite] (9 avec 3)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 1 | 3 | 4 | 2 | 9 | 7 | 10 | 6 | 8 |

gauche 🡨 5

droite 🡨 4

Échanger A[droite] avec pivot (2 avec 5)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 4 | 5 | 9 | 7 | 10 | 6 | 8 |

Pivot (5) a pris sa place. Refaire la même chose pour la partie du tableau située a gauche du pivot et pour la partie du tableau située à droite du pivot.