Université de Liège

Structures de données et algorithmes

Algorithmes

2ème bachelier en ingénieur civil

Auteurs :
Antoine Wehenkel
Antoine Louis
Maxime Lamborelle

Professeur :
P. Geurts

Année académique 2016-2017

Table des matières

1 Introduction et récursivité

1.1 Tri par insertion

1.1.1 Pseudo-code

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

1.1.2 Complexité

En temps

- Dans le pire cas, la boucle **for** est exécutée $(n-1)$ fois et la boucle **while** $(j-1)$ fois. On a alors $T(n) = \Theta(n^2)$.
- Dans le meilleur cas, la boucle **for** est exécutée $(n-1)$ fois et la boucle **while** 0 fois. On a alors $T(n) = \Theta(n)$.
- Dans le cas moyen, cet algorithme est $\Theta(n^2)$.

En espace $O(1)$

1.1.3 Propriétés

Itératif	Récursif	En place	Stable
Oui	Non	Oui	Oui

1.2 Tri par fusion (Merge-Sort)

1.2.1 Pseudo-code

```
MERGE-SORT(A, p, r)
if p < r
    q = (p+r)/2
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
Soit L[1...n1 + 1] et R[1...n2 + 1] deux nouveaux tableaux
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n1
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1 ; j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
```

1.2.2 Complexité

En temps Dans tous les cas, la complexité est $\Theta(n\ log\ n)$.

En espace $O(n)$

1.2.3 Propriétés

Itératif	Récursif	En place	Stable
Oui	Oui	Non	Oui

2 Tri

2.1 Tri rapide (QuickSort)

2.2 Pseudo-code

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
      if A[j] ≤ x
            i = i + 1
            swap(A[i], A[j])
swap(A[i + 1], A[r])
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

Appel initial : QUICKSORT(A,1,A.length)

2.2.1 Complexité

En temps

- Dans le pire cas, q = p ou q = r.On a alors $T(n) = \Theta(n^2)$
- Dans le meilleur cas, q = $\lfloor n/2 \rfloor$. On a alors $T(n) = \Theta(n\ log\ n)$
- Dans le cas moyen, cet algorithme est $\Theta(n\ log\ n)$

Complexité de partition : $T(n) = \Theta(n)$

En espace $O(log\ n)$ si bien implémenté (récursif terminal, en développant d'abord la partition la plus petite)

2.2.2 Propriétés

Itératif	Récursif	En place	Stable
Oui	Oui	Oui	Non

2.3 Construction d'un tas (Build-Max-Heap) et tri par tas (Heap-Sort)

2.3.1 Pseudo-code

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size ∧ A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size ∧ A[r] > A[largest]
    largest = r
if largest ≠ i
    swap(A[i], A[largest])
    MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

```
HEAP-SORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    swap(A[i], A[1])
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

2.3.2 Complexité

- MAX-HEAPIFY a une complexité au pire cas égale à la hauteur du nœud :$T(n) = O(log(n)) = O(h)$. On a donc comme complexité dans le pire cas les cas pour BUILD-MAX-HEAP de $T(n) = \Theta(n)$, cf. slides 154-158.
- On a pour le tri par tas une complexité dans tous les cas égale à $\Theta(n\ log\ n)$ en temps et $O(1)$ en espace.

2.3.3 Propriétés

Itératif	Récursif	En place	Stable
Oui	Oui	Oui	Non

Les points forts du tri par tas sont son efficacité et sa faible consommation de mémoire.

3 Structures de données élémentaires

3.1 Parcours d'arbre

Tous ces algorithme ont une complexité en temps dans tous les cas $\Theta(n)$.

3.1.1 Parcours infixe

Parcours infixe (en ordre) : Chaque nœud est visité après son fils gauche et avant son fils droit.

```
INORDER-TREE-WALK(T, x)
if HASLEFT(T, x)
      INORDER-TREE-WALK(T, LEFT(x))
print GETDATA(T, x)
if HASRIGHT(T, x)
      INORDER-TREE-WALK(T, RIGHT(x))
```

3.1.2 Parcous préfixe

Parcours préfixe (en préordre) : chaque nœud est visité avant ses fils.

```
PREORDER-TREE-WALK(T, x)
print GETDATA(T, x)
if HASLEFT(T, x)
      PREORDER-TREE-WALK(T, LEFT(x))
if HASRIGHT(T, x)
      PREORDER-TREE-WALK(T, RIGHT(x))
```

3.1.3 Parcours postfixe

Parcours postfixe (en postordre) : chaque nœud est visité après ses fils

```
POSTORDER-TREE-WALK(T, x)
if HASLEFT(T, x)
      POSTORDER-TREE-WALK(T, LEFT(x))
if HASRIGHT(T, x)
      POSTORDER-TREE-WALK(T, RIGHT(x))
print GETDATA(T, x)
```

3.1.4 Parcours en largeur

Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite des nœuds de profondeur 1, puis 2, ...

```
BREADTH-TREE-WALK(T)
Q ="Empty queue"
if not ISEMPTY(T)
      ENQUEUE(Q, ROOT(T))
while not QUEUE-EMPTY(Q)
      y = DEQUEUE(Q)
      print GETDATA(T, y)
      if HASLEFT(T, y)
            ENQUEUE(Q, LEFT(y))
      if HASRIGHT(T, y)
            ENQUEUE(Q, RIGHT(y))
```

3.2 Insertion dans un tas (Heap-Insert)

3.2.1 Pseudo-code

```
HEAP-INSERT(A, key)
A.heap − size = A.heap − size + 1
A[A.heap − siez] = −∞
HEAP-INCREASE-KEY(A, A.heap − size, key)
```

```
HEAP-INSERT-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    swap(A[i], A[PARENT(i)])
    i = PARENT(i)
```

3.2.2 Complexité

$O(log\ n)$ car la longueur de la branche de la racine à i est $O(log\ n)$ pour un tas de taille n.

4 Dictionnaires

4.1 Recherche dichotomique

4.1.1 Peudo-code

```
BINARY-SEARCH(V, k, low, high)
if low > high
    return NIL
mid = ⌊(low + high)/2⌋
x = ELEM-AT-RANK(V, mid)
if k == x.key
    return x
elseif k > x.key
    return BINARY-SEARCH(V, k, mid + 1, high)
else return BINARY-SEARCH(V, k, low, mid − 1)
```

4.1.2 Complexité et comparaison avec les autres algorithmes

	Pire cas	*En moyenne*
Implémentation	SEARCH	
Liste	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(\log n)$
ABR	$\Theta(n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$
Table de hachage	$\Theta(n)$	$\Theta(1)$

4.2 Arbre binaire de recherche

		Pire cas			*En moyenne*	
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Table de hachage	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

4.2.1 Recherche

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

Appel initial (à partir d'un arbre T)
TREE-SEARCH($T.root, k$)

Complexité : $T(n) \in O(h)$, où h est la hauteur de l'arbre car dans le pire cas $h = n$.

4.2.2 Successeur

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x == y.right
    x = y
    y = y.parent
return y
```

Complexité : $O(h)$, où h est la hauteur de l'arbre

4.2.3 Insertion

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y == NIL
    // Tree T was empty
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Complexité : $O(h)$ où h est la hauteur de l'arbre

4.2.4 Suppression

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else // z has two children
    y = TREE-SUCCESSOR(z)
    if y.parent ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.parent = y
    // Replace z by y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.parent = y
```

Transplant(T, u, v)

```
if u.parent == NIL
    T.root = v
elseif u == u.parent.left
    u.parent.left = v
else u.parent.right = v
if v ≠ NIL
    v.parent = u.parent
```

Complexité : $O(h)$ pour un arbre de hauteur h
(Tout est $O(1)$ sauf l'appel à Tree-Successor).

5 Graphes

5.1 Parcours en largeur

5.1.1 Pseudo-code

```
BFS(G, s)
for each vertex u ∈ G.V \ {s}
    u.d = ∞
s.d = 0
Q = "create empty Queue"
ENQUEUE(Q, s)
while not QUEUE-EMPTY(Q)
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.d = ∞
            v.d = u.d + 1
            ENQUEUE(Q, v)
```

5.1.2 Complexité

- Chaque sommet est enfilé au plus une fois ($v.d$ infini $\rightarrow$ fini)
- Boucle exécutée $O(|V|)$ fois
- Boucle interne $O(|E|)$ fois au total
- Au total : $O(|V| + |E|)$

5.2 Parcours en profondeur

5.2.1 Pseudo-code

```
DFS(G, s)
for each vertex u ∈ G.V
    u.visited = FALSE
S = "create empty Stack"
PUSH(S, s)
while not STACK-EMPTY(S)
    u = POP(S)
    if u.visited == FALSE
        u.visited == TRUE
        for each v ∈ G.Adj[u]
            if v.visited == FALSE
                PUSH(S, v)
```

5.2.2 Complexité

- Initialisation : $\Theta(|V|)$
- Boucle **while** $O(|V| + |E|)$
- Au total : $O(|V| + |E|)$

5.3 Bellman-Ford

5.3.1 Pseudo-code

BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ **to** $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX(u, v, w)

INIT-SINGLE-SOURCE(G, s)

for each $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

Complexité : $\Theta(|V| \cdot |E|)$

5.4 Dijkstra

5.4.1 Pseudo-code

DIJKSTRA(G, w, s)

INIT-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q =$ "create an empty min priority queue from $G.V$"
while not EMPTYQ
    $u =$ EXTRACT-MIN(Q)
    $S = S \bigcup \{u\}$
    for each $v \in G.Adj[u]$
        RELAX(u, v, w) // ! RELAX doit modifier la clé de v dans Q

5.4.2 Complexité

- Si la file est un tas extraction et ajustement de clé en : $O(\log(|V|))$
- Chaque sommet est extrait de la file à priorité une et une seule fois : $O(|V| \cdot \log(|V|))$
- Chaque arête est parcourue une et une seule fois et entraîne au plus un ajustement de clé : $O(|E| \cdot \log(|V|))$
- Total : $O(|V| \cdot \log(|V|) + |E| \cdot \log(|V|)) = O(|E| \cdot \log(|V|))$

5.5 Kruskal

5.5.1 Pseudo-code

```
KRUSKAL(G, w)
A = ∅
P = ∅
for each vertex v ∈ G.V
    P = P ⋃ {{v}}
for each (u, v) ∈ G.E taken in nondecreasing order of weight w
    P1 = subset in P containing u
    P2 = subset in P containing v
    if P1 ≠ P2
        A = A ⋃ {(u, v)}
        Merge P1, P2 et P
return A
```

5.5.2 Complexité

- Initialisation : $O(|V|)$
- Tri des arêtes : $O(|E| \cdot \log(|V|))$
- Coût total des fusions : $O(|V| \cdot \log(|V|))$
- Total : $O(|V| \cdot \log(|V|) + |E| \cdot \log(|V|)) = O(|E| \cdot \log(|V|))$

5.6 Prim

5.6.1 Pseudo-code

```
PRIM(G, w, r)
Q = ∅
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
    INSERT(Q, u)
DECREASE-KEY(Q, r, 0) // r.key = 0
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            DECREASE-KEY(Q, v, w(u, v))
```

5.6.2 Complexité

- Initialisation et première boucle **for** : $O(|V| \cdot \log(|V|))$
- Diminuer la clé de r : $O(\log(|V|))$
- Boucle **while** ($|V|$ appels à EXTRACT-MIN et $|E|$ appels à DECREASE-KEY) : $O(|V| \cdot \log(|V|) + |E| \cdot \log(|V|))$
- Total : $O(|V| \cdot \log(|V|) + |E| \cdot \log(|V|)) = O(|E| \cdot \log(|V|))$