Solutions du CLRS 3 *ed.*

Exercises : (81/29/957)
Problems : (0/6/158)

22 septembre 2017

Table des matières

Liste des Algorithmes

1 The Role of Algorithms in Computing

1.1 Algorithms

1.1-1 *Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.*

Solution :

— Tri :

- Tri de cartes ;
- Tri des prix en *e-shopping*.

— Enveloppe convexe : « The problem of finding convex hulls finds its practical applications in pattern recognition, image processing, statistics, geographic information system, game theory, construction of phase diagrams, and static code analysis by abstract interpretation. It also serves as a tool, a building block for a number of other computational-geometric algorithms such as the rotating calipers method for computing the width and diameter of a point set. » [2]

■

1.1-2 *Other than speed, what other measures of efficiency might one use in a real-world setting ?*

Solution : La mémoire (l'espace) et l'utilisation de ressources. ■

1.1-3 *Select a data structure that you have seen previously, and discuss its strengths and limitations.*

Solution : Un simple exemple serait la comparaison de liste chaînée et de tableau :

— liste chaînée :

✓ la taille de liste chaînée peut varier suivant le temps ;

✓ l'insertion et la supression d'éléments de la liste peuvent être réalisés facilement ;

✗ requiert davatange mémoire pour stocker les références ;

✗ accès séquentiel d'élément.

— tableau :

✓ accès aléatoire d'élément ;

✗ la taille du tableau est fixée dès sa déclaration ;

✗ l'insertion et la suppression d'élément requièrent une réallocation du tableau, ce qui est coûteux.

■

1.1-4 *How are the shortest-path and traveling-salesman problems given above similar ? How are they different ?*

Solution : La similitude des deux problèmes est la recherche d'un parcours sur un graphe. La différence est que le TSP contient plus de contraintes : on cherche un cycle hamiltonien suffisamment court (ou le optimal si possible) parmis les nœuds donnés tandis qu'au *shortest-path problem* on cherche le plus court parcours entre deux nœuds. ■

1.1-5 *Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.*

Solution : Cela est l'étude centrale de l'analyse numérique. On se contente d'avoir des résultats approximatifs avec une précision satisfaisante. ■

1.2 Algorithms as a technology

1.2-1 *Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.*

Solution :

■

1.2-2 *Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n\lg n$ steps. For which values of n does insertion sort beat merge sort?*

Solution : On cherche $N \in \mathbb{N}$ tel que $8N^2 < 64N\lg N$, ce qui revient à trouver les $N > 1$ qui vérifie $\frac{N}{\lg N} < 8$. D'après Wolfram Alpha, $N < 43.56$, donc lorsque $N \in [\![1, 43]\!]$, le tri d'insertion est plus plus efficace que le tri fusion. ■

1.2-3 *What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?*

Solution : On cherche l'entier obtenu par la solution de $100n^2 = 2^n$ arrondi par excès. D'après Wolfram Alpha on trouve que $n > 14.324$, donc $n = 15$. ■

1.3 Problems

1-1 ***Comparison of running times***

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

Solution :

■

2 Getting Started

2.1 Insertion Sort

2.1-1 *Using Figure 2.2 as a model, illustrate the operation of* Insertion-Sort *on the array* $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

Solution :

■

2.1-2 *Rewrite the* Insertion-Sort *procedure to sort into non-increasing instead of non-decreasing order.*

Solution :

Algorithme 1. Decr-Insertion-Sort (A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] < key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

■

2.1-3 *Consider the* ***searching problem*** *:*

Input : *A sequence of* n *numbers* $A = \langle a_1, a_2, \ldots, a_n \rangle$ *and a value* v.

Output : *An index* i *such that* $v = A[i]$ *or the special value* NIL *if* v *does not appear in* A.

Write pseudocode for ***linear search****, which scans through the sequence, looking for* v. *Using a loop, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.*

Solution :

Algorithme 2. Linear-Search (A, v)

```
1  i = 1
2  while i ≤ A.length and A[i] ≠ v
3      i = i + 1
4  if i ≤ A.length
5      return i
6  else
7      return NIL
```

— Initialisation : $i = 1$

$[1..i-1]$ est vide, donc ne contient pas v.

— Conservation

Pour tout $1 \leq i \leq \mathrm{A}.length$, pour la boucle i, on a $[1..i-1]$ ne contenant pas v. Le corps de la boucle est executé si et seulement si $\mathrm{A}[i] \neq v$ et que $i \leq \mathrm{A}.length$. Dans ce cas, avant l'itération $i+1$, le sous-tableau $[1..i]$ ne contient pas v.

— Terminaison

Il y a deux cas de terminaison :

- quand $i = \mathrm{A}.length + 1$, l'algorithme retourne NIL et l'invariant de boucle confirme (en subtituant i par $\mathrm{A}.length + 1$) que le tableau $[1..\mathrm{A}.length]$ ne contient v ;
- il existe un $1 \leq i \leq \mathrm{A}.length$ tel que $\mathrm{A}[i] = v$, l'invariant de boucle dit que $[1..i-1]$ ne contient pas v, ce qui est vraie. Dans ce cas l'algorithme retourne i.

■

2.1-4 *Consider the problem of adding two n-bit binary integers, stored in two n-element arrays* A *and* B. *The sum of the two integers should be stored in binary form in an* $(n+1)$*-element array* C. *State the problem formally and write pseudocode for adding the two integers.*

Solution :

Input : Deux nombres binaires a et b sous forme de vecteur $\mathrm{A} = \langle a_1, \ldots, a_n \rangle$ et $\mathrm{B} = \langle b_1, \ldots, b_n \rangle$ tel que pour tout $i \in [\![1, n]\!], a_i, b_i \in \{0, 1\}$. Avec l'indice $i = 1$ désignant le bit le plus significant.

Output : Le vecteur $\mathrm{C} = \langle c_1, \ldots, c_{n+1} \rangle$ qui représente $c = a + b$ en binaire.

Algorithme 3. ADD-BINARY-INTEGER (A, B)

```
1  carry = 0
2  for i = n downto 1
3       tmp = A[i] + B[i] + carry
4       C[i + 1] = tmp  mod 2
5       carry = tmp/2
6  C[i] = carry
```

■

2.2 Analyzing algorithms

2.2-1 *Express the function* $n^3/1000 - 100n^2 - 100n + 3$ *in terms of* Θ*-notation.*

Solution : $\Theta(n^3)$ ■

2.2-2 *Consider sorting n numbers stored in array* A *by first finding the smallest element of* A *and exchanging it with the element in* A[1]. *Then find the second smallest element of* A, *and exchange it with* A[2]. *Continue in this manner for the first* $n-1$ *elements of* A. *Write pseudocode for this algorithm, which is known as* ***selection sort***. *What loop invariant does this algorithm maintain? Why does it need to run for only the first* $n-1$ *elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in* Θ*-notation.*

Solution :

Algorithme 4. SELECTION-SORT (A)

```
1  for i = 1 to n − 1
2        min = i
3        for j = i + 1 to n
4              if A[min] > A[j]
5                    min = j
6        if min ≠ j
7              swap(A, min, j)
```

— Invariant de boucle :

Le tableau $[1..i-1]$ est trié avant la $i$ème itération et tous les éléments dans $[i..n]$ sont supérieurs à ceux dans $[1..i-1]$.

— À la $n-1$ itération, le tableau $[1..n-2]$ est trié, il ne reste plus qu'à comparer $A[n-1]$ et $A[n]$.

— Temps d'exécution :

- Cas optimal : A déjà trié, $\Theta(n^2)$;
- Cas le plus défavorable : A trié de façon décroissante, $\Theta(n^2)$.

■

2.2-3 *Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in* Θ*-notation? Justify your answers.*

Solution :

— Cas moyen : $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$;

— Cas le plus défavorable : $n+1$.

Donc $\Theta(n)$ dans les deux cas. ■

2.2-4 *How can we modify almost any algorithm to have a good best-case running time?*

Solution : Tester au début de l'algorithme la nature de l'entrée, si celle-ci vérifie une certaine condition, retourner un résultat calculé préalablement. ■

2.3 Designing algorithms

2.3-1 *Using Figure 2.4 as a model, illustrate the operation of merge sort on the array* $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

Solution :

■

2.3-2 *Rewrite the* MERGE *procedure so that it does not use sentinels, instead stopping once either array* L *or* R *has had all its elements copied back to* A *and then copying the remainder of the other array back into* A.

Solution :

Algorithme 5. MERGE (A, p, q, r)

```
1  n1 = q − p + 1
2  n2 = r − q
3  Let L[1..n1] and R[1..n2] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  i = 1
9  j = 1
10 for k = p to r
11     if j == n2 + 1 or (i ≤ n1 and L[i] ≤ R[j])
12         A[k] = L[i]
13         i = i + 1
14     else
15         A[k] = R[j]
16         j = j + 1
```

■

2.3-3 *Use mathematical induction to show that when* n *is an exact power of* 2, *the solution of the recurrence*

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

Solution : Soit $n = 2^p$, avec $p \in \mathbb{N}^*$.

— Initialisation : $T(2) = 2 \lg 2 = 2$ est vraie.

— Hérédité : Soit $k < p$ et supposons que $T(2^k) = 2^k \lg 2^k = 2^k k$.
Alors $T(2^{k+1}) = 2T(2^k) + 2^{k+1} = 2^{k+1}(k+1)$.

— Conclusion : Pour tout $p \in \mathbb{N}^*$, on a bien $T(2^p) = 2^p p$.

■

2.3-4 *We can express insertion sort as a recursive procedure as follows. In order to sort* $A = [1..n]$, *we recursively sort* $A = [1..n-1]$ *and then insert* $A[n]$ *into the sorted array* $A[1..n-1]$ *Write a recurrence for the running time of this recursive version of insertion sort.*

Solution :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}$$

■

2.3-5 *Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence* A *is sorted, we can check the midpoint of the sequence against and eliminate half of the sequence from further consideration. The* ***binary search*** *algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is* $\Theta(\lg n)$.

Solution :

Algorithme 6. Rec-Bin-Search (A, p, q, v)

```
1  mid = ⌊(p + q)/2⌋
2  if A[mid] < v
3        Rec-Bin-Search(A, r + 1, q, v)
4  else if A[mid] > v
5        Rec-Bin-Search(A, p, mid − 1, v)
6  else if A[mid] == v
7        return mid
8  else
9        return NIL
```

Calculons le temps d'exécution au cas le plus défavorable. On a :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{otherwise} \end{cases}.$$

Supposons que $n = 2^p$, alors par la méthode d'arbre récursive, on a un arbre dégénéré de hauteur p dans lequel chaque nœud est étiqueté par une constante c. Il suffit donc

de calculer

$$\begin{aligned}\sum_{i=0}^{p} c &= (p+1)c \\ &= (\lg n + 1)c \\ &= \Theta(\lg n).\end{aligned}$$

■

2.3-6 *Observe that the* ***while*** *loop of lines* 5–7 *of the* Insertion-Sort *procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray* $A[1..j-1]$. *Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to* $\Theta(n \lg n)$*?*

Solution : Non. Dans la boucle **tant que**, on effectue deux tâches :

1. la recherche linéaire de l'emplacement auquel on insère l'élément : $\Theta(n)$;
2. le décalage de tous les éléments après cet emplacement : $\Theta(n)$.

Remplacer la recherche linéaire par la recherche dichotomique améliore la première tâche en temps logarithmique. Néanmoins, le décalage toujours en $\Theta(n)$, cause un temps quadratique à l'algorithme. ■

2.3-7 ⋆ *Describe a* $\Theta(n \lg n)$*-time algorithm that, given a set* S *of* n *integers and another integer* x*, determines whether or not there exist two elements in* S *whose sum is exactly* x.

Solution :

■

2.4 Problems

2-1 ***Insertion sort on small arrays in merge sort***

Although merge sort runs in $\Theta(n \lg n)$ *worst-case time and insertion sort runs in* $\Theta(n^2)$ *worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to* ***coarsen*** *the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which* $n = k$ *sublists of length* k *are sorted using insertion sort and then merged using the standard merging mechanism, where* k *is a value to be determined.*

a. *Show that insertion sort can sort the* $n = k$ *sublists, each of length* k*, in* $\Theta(nk)$ *worst-case time.*

b. *Show how to merge the sublists in* $\Theta(n \lg(n/k))$ *worst-case time.*

c. *Given that the modified algorithm runs in $\Theta(nk + n\lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?*

d. *How should we choose k in practice?*

Solution :

■

2-2 *Correctness of bubblesort*

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

Solution :

■

2-3 *Correctness of Horner's rule*

The following code fragment implements Horner's rule for evaluating a polynomial

Solution :

■

2-4 *Inversions*

Solution :

■

3 Growth of Functions

3.1 Asymptotic notation

3.2 Standard notations and common functions

4 Divide-and-Conquer

4.1 The maximum-subarray problem

4.1-1 *What does* Find-Maximum-Subarray *return when all elements of A are negative?*

Solution : L'indice du plus grand élément du tableau. ■

4.1-2 *Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in* $\Theta(n^2)$ *time.*

Solution :

Algorithme 7. Brute-Find-Max-Subarray (A)

```
1  index = period = −1
2  sum = −∞
3  for i = 1 to A.length
4      tmp-sum = 0
5      for tmp-period = 1 to A.length − i + 1
6          tmp-sum = tmp-sum + A[i + tmp-period − 1]
7          if tmp-sum > sum
8              index = i
9              period = tmp-period
10             sum = tmp-sum
11 return (index, period, sum)
```

■

4.1-3 *Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size* n_0 *gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than* n_0*. Does that change the crossover point?*

Solution :

— Environ 17, voir le programme `tst_crossover_pt_1`.

— Le *crossover point* est devenu 2, voir le programme `tst_crossover_pt_2`.

■

4.1-4 *Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subarray is* 0*. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?*

Solution :

— Si la somme finale est négative, on retourne 0 et le tableau vide ;

— Initialiser par 0 la somme.

■

4.1-5 *Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of* $A[1..j]$*, extend the answer to find a maximum subarray ending at index* $j+1$ *by using the following observation : a maximum subarray of* $A[1..j+1]$ *is either a maximum subarray of* $A[1..j]$ *or a subarray* $A[i..j+1]$*, for some* $1 \le i \le j+1$*. Determine a maximum subarray of the form* $A[i..j+1]$ *in constant time based on knowing a maximum subarray ending at index* j*.*

Solution :

Algorithme 8. LINEAR-FIND-MAX-SUBARRAY (A)

```
1  left = right = 1
2  sum = A[1]
3  for j = 2 to n
4       tmp-sum = 0
5       if A[j] ≤ 0
6            continue
7       else if right == j − 1
8            right = j
9            sum = sum + A[j]
10      else
11           right = j
12           i = j − 1
13           while A[i] > 0
14                i = i − 1
15                left = i + 1
16 return sum
```

■

4.2 Strassen's algorithm for matrix multiplication

4.2-1 *Use Strassen's algorithm to compute the matrix product*

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

Show your work.

Solution : Soit $A = \begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}$ et $B = \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$, on veut calculer le produit matriciel $AB = C$.
On *divise* les matrices A et B en 4 sous-matrices :

$$\begin{array}{llll} A_{11} = (1) & A_{12} = (3) & \text{et} & B_{11} = 6 \quad B_{12} = 8 \\ A_{21} = (7) & A_{22} = (5) & & B_{21} = 4 \quad B_{22} = 2 \end{array}$$

puis on effectue les calcules intermédiaires :

$$\begin{array}{ll} S_1 = B_{12} - B_{22} = 6 & S_6 = B_{11} + B_{22} = 8 \\ S_2 = A_{11} + A_{12} = 4 & S_7 = A_{12} - A_{22} = -2 \\ S_3 = A_{21} + A_{22} = 12 & S_8 = B_{21} + B_{22} = 6 \\ S_4 = B_{21} - B_{11} = -2 & S_9 = A_{11} - A_{21} = -6 \\ S_5 = A_{11} + A_{22} = 6 & S_{10} = B_{11} + B_{12} = 14 \end{array}$$

et

$$\begin{array}{ll} P_1 = A_{11} \cdot S_1 = 6 & P_4 = A_{22} \cdot S_4 = -10 \\ P_2 = S_2 \cdot B_{22} = 8 & P_5 = S_5 \cdot S_6 = 48 \\ P_3 = S_3 \cdot B_{11} = 72 & P_6 = S_7 \cdot S_8 = -12 \\ & P_7 = S_9 \cdot S_{10} = -84 \end{array}$$

On finie par :

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 = 18 \\ C_{12} &= P_1 + P_2 = 14 \\ C_{21} &= P_3 + P_4 = 62 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 = 66 \end{aligned}$$

D'où

$$C = \begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}.$$

■

4.2-2 *Write pseudocode for Strassen's algorithm.*

Solution :

■

4.2-3 *How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.*

Solution :

■

4.2-4 *What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?*

Solution :

■

4.2-5 *V. Pan has discovered a way of multiplying 68×68 matrices using $132,464$ multiplications, a way of multiplying 70×70 matrices using $143,640$ multiplications, and a way of multiplying 72×72 matrices using $155,424$ multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare to Strassen's algorithm?*

Solution :

■

4.2-6 *How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.*

Solution :

— $kn \times n : \Theta(k^2 n^{\lg 7})$

— $n \times kn : \Theta(kn^{\lg 7})$

■

4.2-7 *Show how to multiply the complex numbers $a+bi$ and $c+di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.*

Solution :

Algorithme 9. ADD-COMPLEX-NUMBER (a, b, c, d)

1 $P_1 = ac$
2 $P_2 = bd$
3 $P_3 = (a + b)(c + d)$
4 $real = P_1 - P_2$
5 $imag = P_3 - P_1 - P_2$
6 **return** $(real, imag)$

Ceci est un cas particulier de l'algortihme de Karatsuba [1] qui a une complexité en temps $\mathcal{O}(n^{\lg 3})$ pour la multiplication entre deux nombres de n chiffres.

■

4.3 The substitution method for solving recurrences

4.3-1 *Show that the solution of* $\mathrm{T}(n) = \mathrm{T}(n-1) + n$ *is* $\mathcal{O}(n^2)$.

Solution : Supposons que $\mathrm{T}(n) \leq cn^2$. On a

$$\begin{aligned}
\mathrm{T}(n) &\leq c(n-1)^2 + n \\
&= c(n^2 - 2n + 1) + n \\
&= cn^2 - 2cn + c + n \\
&\leq cn^2 \qquad \text{avec } c \geq \frac{n}{2n-1}.
\end{aligned}$$

La fonction $\frac{n}{2n-1}$ étant décroissante, on peut choisir $c = 1$ et $n_0 = 1$ tel que la définition soit vérifiée :

$$\mathrm{T}(n) \in \left\{ f(n) : \forall n \geq 1,\ 0 \leq f(n) \leq n^2 \right\} \Longrightarrow \mathrm{T}(n) \in \mathcal{O}(n^2).$$

■

4.3-2 *Show that the solution of* $\mathrm{T}(n) = \mathrm{T}(\lceil n/2 \rceil) + 1$ *is* $\mathcal{O}(\lg n)$.

Solution : Supposons que $\mathrm{T}(n) \leq c \lg n$ est vraie pour tout $m \leq n$ en particulier $m = \lceil n/2 \rceil$. Alors

$$\begin{aligned}
\mathrm{T}(n) &\leq c \lg n/2 + 1 \\
&= c(\lg n - 1) + 1 \\
&= c \lg n - c + 1 \\
&\leq c \lg n \qquad \forall c \in]0, 1].
\end{aligned}$$

■

4.3-3 *We saw that the solution of* $\mathrm{T}(n) = 2\mathrm{T}(\lfloor n/2 \rfloor) + n$ *is* $\mathcal{O}(n \lg n)$. *Show that the solution of this recurrence is also* $\Omega(n \lg n)$. *Conclude that the solution is* $\Theta(n \lg n)$.

1. voir Algorithme de Karatsuba (Wikipedia)

Solution : On suppose que $T(n) \geq c(n+2)\lg(n+2)$. On a

$$\begin{aligned} T(n) &\geq 2c(\lfloor n/2 \rfloor + 2)\lg(\lfloor n/2 \rfloor + 2) + n \\ &\geq 2c(n/2+1)\lg(n/2+1) + n \\ &= c(n+2)(\lg(n+2) - 1) + n \\ &= c(n+2)\lg(n+2) - c(n+2) + n \\ &\geq c(n+2)\lg(n+2) \qquad \text{avec } c \leq 1/3. \end{aligned}$$

De plus, $c(n+2)\lg(n+2) \geq cn\lg n$, donc on a bien $T(n) = \Omega(n\lg n)$ pour $c = 1/3$ et $n_0 = 1$ par exemple.

■

Solution :

■

4.3-4

Solution :

■

4.3-5

Solution :

■

4.3-6

Solution :

■

4.3-7

Solution :

■

4.3-8

Solution :

■

4.3-9

4.4 The recursion-tree method for solving recurrences

4.5 The master method for solving recurrences

4.5-1 *Use the master method to give tight asymptotic bounds for the following recurrences.*

a. $T(n) = 2T(n/4) + 1$.

b. $T(n) = 2T(n/4) + \sqrt{n}$.

c. $T(n) = 2T(n/4) + n$.

d. $T(n) = 2T(n/4) + n^2$.

Solution : Dans les quatre cas, on a $\log_b a = \log_4 2 = \frac{1}{2}$.

a. $f(n) = 1 = \mathcal{O}(n^{\log_4 2 - \varepsilon})$ avec $\varepsilon \in]0, 1]$, donc $T(n) = \Theta(\sqrt{n})$.

b. $f(n) = \sqrt{n} = \Theta(\sqrt{n})$, donc $T(n) = \Theta(\sqrt{n} \lg n)$.

c. $f(n) = n = \Omega(n^{\log_4 2 + \varepsilon})$ avec $\varepsilon \in]0, 2]$, de plus $af(n/b) = n/2 \leq cf(n) = cn$ avec $1/2 \leq c < 1$, donc $T(n) = \Theta(n)$.

d. $f(n) = n^2 = \Omega(n^{\log_4 2 + \varepsilon})$ avec $\varepsilon \in]0, 14]$, de plus $af(n/b) = n^2/8 \leq cf(n)$ avec $1/8 \leq c < 1$, donc $T(n) = \Theta(n^2)$.

■

4.5-2 *Professor Caesar wishes to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into pieces of size $n/4 \times n/4$, and the divide and combine steps together will take $\Theta(n^2)$ time. He needs to determine how many subproblems his algorithm has to create in order to beat Strassen's algorithm. If his algorithm creates a subproblems, then the recurrence for the running time $T(n)$ becomes $T(n) = aT(n/4) + \Theta(n^2)$. What is the largest integer value of a for which Professor Caesar's algorithm would be asymptotically faster than Strassen's algorithm?*

Solution :

■

4.5-3 *Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$. (See Exercise 2.3-5 for a description of binary search.)*

Solution : On a $a = 1, b = 2$, alors $n^{\lg_b a} = 1$. Comme $f(n) \in \Theta(1) = \Theta(n^{\lg_b a})$, $T(n) = \Theta(\lg n)$. ■

4.5-4 *Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \lg n$? Why or why not? Give an asymptotic upper bound for this recurrence.*

Solution : On compare $n^{\lg_2 4} = n^2$ avec $f(n) = n^2 \lg n$. Comme on n'a ni $f(n) = \mathcal{O}(n^{\lg_2 4-\varepsilon})$ ni $f(n) = \Theta(n^2)$ ni $f(n) = \Omega(n^{\lg_2 4+\varepsilon})$, on ne peut utiliser la *master method*. Pour trouver une borne supérieure asymptotique, on peut d'abord conjecturer avec la méthode d'arbre récursive puis vérifier l'exactitude avec la méthode de substitution.

Arbre récursive : Supposont que $n = 2^p$ avec $p \in \mathbb{N}$. On a un arbre complet de hauteur p dans lequel chaque nœud est étiqueté par $cn^2 \lg n$ et a 4 fils. Chaque niveau $i \in \{0, \dots, p-1\}$ a un coût $c4^i \cdot c(\frac{n}{2^i})^2 \lg \frac{n}{2^i} = cn^2(p-i)$. Les feuilles sont supposées d'être étiquetées par une constante $\Theta(1)$ et il y en a $4^p = n^2$, donc le coût des feuilles est $\Theta(n^2)$. On obtient, en sommant les coûts de tous les niveaux, une idée sur la borne asymptotique : za

$$\begin{aligned}\sum_{i=0}^{p-1} cn^2(p-k) + \Theta(n^2) &= cn^2 \sum_{i=0}^{p-1} (p-k) + \Theta(n^2) \\ &= cn^2(p^2 - \frac{(p-1)p}{2}) \\ &= \Theta(n^2 \lg^2 n) = \Theta\left((n \lg n)^2\right).\end{aligned}$$

Méthode de substitution : ■

4.5-5 ★ *Consider the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of constant $a \geq 1$ and $b > 1$ and a function $f(n)$ that satisfies all the conditions in case 3 of the master theorem except the regularity condition.*

Solution : Fixons $a = 1, b = 2$ et étudions les fonctions de forme $f(n) = a_n \cdot n$ avec $a_n > 0$ qui pourraient vérifier

$$\begin{cases} f(n) & = \Omega(n^{\log_2(1+\varepsilon)}) \quad \text{avec } \varepsilon > 0 \\ f(n/2) & \leq cf(n) \quad \text{avec } c \geq 1. \end{cases}$$

La première condition est vérifiée si a_n soit bornée. Pour que la deuxième l'est également, il faut que $a_{n/2} \leq 2ca_n$ et que a_n ne soit pas constante. Intuitivement, on choisit $a_n = \cos(n) + 2$ qui est toujours positive. Par substitution, on obtient l'inégalité suivante

$$\frac{\cos(n/2) + 2}{2(\cos(n) + 2)} \leq c \tag{1}$$

Le maximum du premier membre de l'inégalité (1) est donné par Wolfram Alpha et vaut approximativement 1.0303. Donc on a $1.0303 \leq c$, ce qui vérifie la deuxième condition. Ainsi, on ne peut appliquer le *master theorem* pour trouver une borne asymptotique inférieure. ■

5 Probabilistic Analysis and Randomized Algorithms

5.1 The hiring problem

5.1-1 *Show that the assumption that we are always able to determine which candidate is best, in line 4 of procedure* Hire-Assistant, *implies that we know a total order on the ranks of the candidates.*

Solution : Les axiomes de l'ordre (transitivité, antisymétrie et reflexivité) ne suffisent pas, la totalité ($x \leq y$ ou $y \leq x$) est exigé. Sinon il se peut qu'on ne pourrait comparer deux candidates. ■

5.1-2 ⋆ *Describe an implementation of the procedure* Random(a, b) *that only makes calls to* Random$(0, 1)$. *What is the expected running time of your procedure, as a function of a and b ?*

Solution :

Algorithme 10. Random (a, b)

```
1  n = b − a
2  c = ⌈lg(n + 1)⌉
3  r = n
4  while r > n
5        r = 0
6        // Form a c-digit binary number
7        for i = 1 to c
8              r = r + Shift-Left(Random(0, 1), i)
9  return a + r
```

— Exactitude : La probabilité d'obtenir un nombre $a + i$ avec $i \in [\![0, n]\!]$ est :

$$\begin{aligned}\mathbb{P}(\{a+i\}) &= \sum_{i=0}^{\infty} \left(\frac{c-n-1}{c}\right)^i \cdot \left(\frac{1}{c}\right) \\ &= \left(\frac{1}{c}\right) \cdot \left(\frac{c}{n+1}\right) \\ &= \frac{1}{n+1}\end{aligned}$$

On a bien une distribution uniforme.

— Complexité en temps : $\mathcal{O}(\lg(b-a))$.

■

5.1-3 ⋆ *Suppose that you want to output 0 with probability 1/2 and 1 with probability 1/2. At your disposal is a procedure* Biased-Random, *that outputs either 0 or 1. It outputs 1 with some probability p and 0 with probability $1/p$, where $0 < p < 1$, but you do not know what*

p is. Give an algorithm that uses Biased-Random *as a subroutine, and returns an unbiased answer, returning* 0 *with probability* 1/2 *and* 1 *with probability* 1/2. *What is the expected running time of your algorithm as a function of p ?*

Solution : On énonce ici le *skew-correction algorithm* dû à von Neumann :

Algorithme 11. Unbiased-Random $(0,1)$

```
1  (a_1, a_2) = (0, 0)
2  while a_1 == a_2
3        (a_1, a_2) = (Biased-Random(0,1), Biased-Random(0,1))
4  return a_1
```

— Exactitude : La probabilité d'obtenir un nombre i avec $i \in \{0,1\}$ est :

$$\sum_{i=0}^{\infty} \left((1-p)^2 + p^2\right)^i p(1-p) = \frac{p(1-p)}{1-(2p^2-2p+1)} = \frac{1}{2}.$$

— Complexité en temps : $\mathcal{O}\left(\frac{1}{p(1-p)}\right)$.

■

Première partie

Sorting and Order Statistics

6 Heapsort

6.1 Heaps

6.1-1 *What are the minimum and maximum numbers of elements in a heap of height h?*

Solution :

— Minimum : 2^h ;

— Maximum : $2^{h+1} - 1$.

■

6.1-2 *Show that an n-element heap has height $\lfloor \lg n \rfloor$.*

Solution : On reprend le résultat précédent (exercice 6.1-1) qui nous offre l'inégalité :

$$2^h \leq n \leq 2^{h+1} - 1.$$

Ainsi, étant donné un nombre d'éléments n, l'hauteur h d'un tas est bornée par :

$$\lg n - 1 < \lg(n+1) - 1 \leq h \leq \lg n.$$

Sachant que h est un entier, à partir de $\lg n - 1 < h \leq \lg n$ on déduit que $h = \lfloor \lg n \rfloor$. ■

6.1-3 *Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.*

Solution : C'est une conséquence immédiate de la propriété d'un *max-heap* : la clé de la racine d'un sous-arbre est supérieur égale à celle de ses enfants ; or, ses enfants sont également la racine du sous-arbre qu'ils apartiennent. De ce fait, une preuve plus complète serait achevée par induction mathématiques. ■

6.1-4 *Where in a max-heap might the smallest element reside, assuming that all elements are distinct?*

Solution : La propriété d'un *max-heap* est que $A[\textsc{Parent}(i)] \geq A[i]$ ce qui signifit que tout chemin partant de la racine forme une suite décroissante, et strictement, par hypothèse. Par conséquence, le plus petit élément réside dans une des feuilles. Ce résultat découle du fait que la relation d'ordre n'est pas totale sur le tas. ■

6.1-5 *Is an array that is in sorted order a min-heap?*

Solution : Si le tableau est trié de manière croissante, le tableau forme un *min-heap*.

■

6.1-6 *Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?*

Solution : Considérons la figure (1) qui est la représentation arborescente du tableau. On voit qu'il existe une sous-suite qui n'est pas décroissante : $(23, 17, 6, 7)$, donc ce tableau ne vérifie pas la propriété d'un *max-heap*.

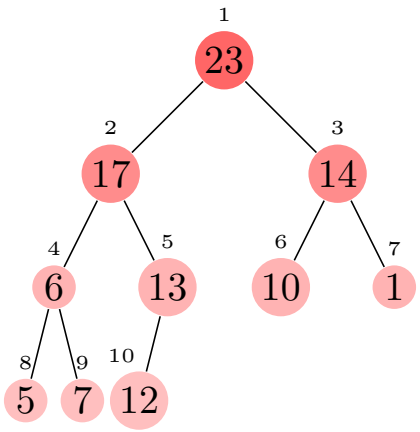

FIGURE 1 – Tas sous forme d'arbre du tableau $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12\rangle$

■

6.1-7 *Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by* $\lfloor n/2\rfloor + 1, \lfloor n/2\rfloor + 2, \ldots, n$.

Solution : Une feuille n'a pas de fils, et comme les nœux sont insérés de façon contigüe, il suffit de trouver le premier k tel que $\textsc{Left}(k) = 2k > n$. On a donc $n/2 < k$ et puis comme k est un entier, il vaut $\lfloor n/2\rfloor + 1$. ■

6.2 Maintaining the heap property

6.2-1 *Using Figure 6.2 as a model, illustrate the operation of* $\textsc{Max-Heapify}(A, 3)$ *on the array* $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$

Solution :

Regardons la figure (2) qui illustre les opérations de $\textsc{Max-Heapify}(A, 3)$: (a) La configuration initiale du tas, avec la valeur $A[3]$ au nœud $i = 3$ viole la propriété de *max-heap* puisque'elle n'est pas supérieur à celle des enfants. La propriété de *max-heap* est restaurée pour le nœud 3 en (b) via échange de $A[3]$ avec $A[6]$, ce qui détruit la propriété de *max-heap* pour le nœud 6. L'appel récursif $\textsc{Max-Heapify}(A, 6)$ prend maintenant $i = 6$. Après avoir échangé $A[6]$ avec $A[13]$, comme illustré en (c), le nœud 6 est corrigé, et l'appel récursif $\textsc{Max-Heapify}(A, 13)$ n'engendre plus de modifications de la structure de données.

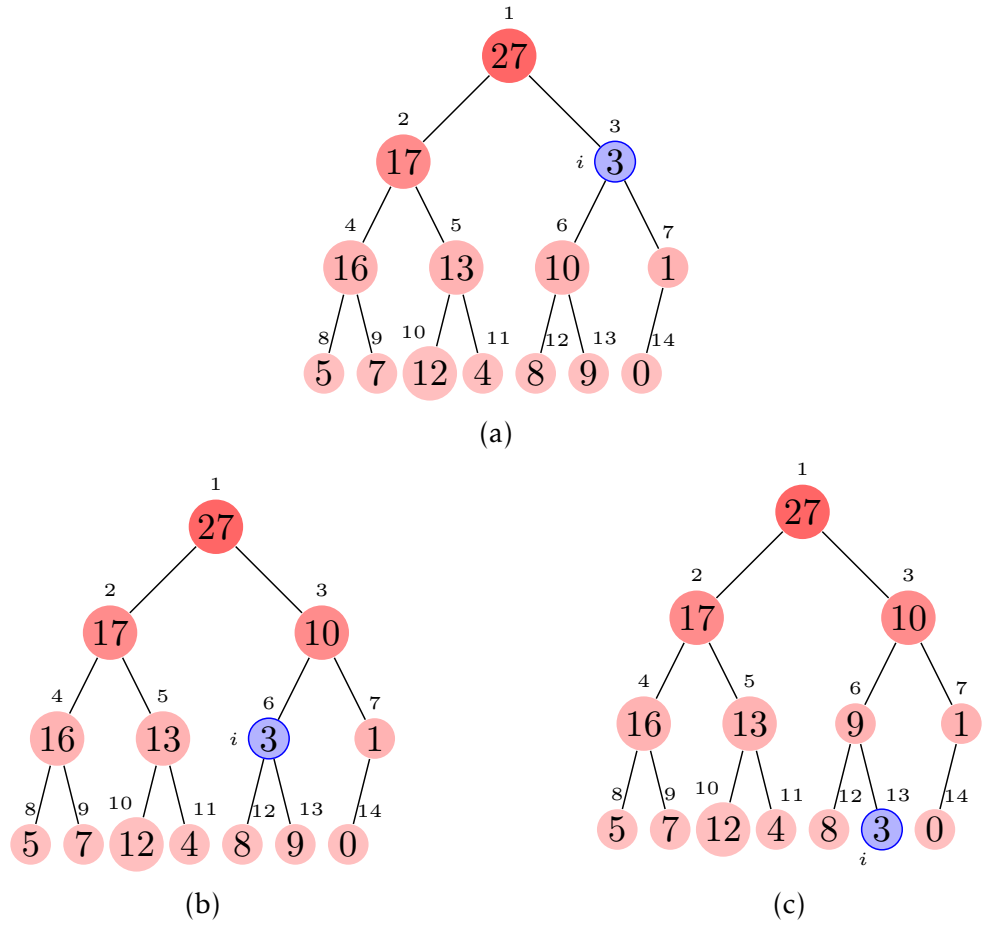

FIGURE 2 – L'action MAX-HEAPIFY(A, 3),où A.*heap-size* = 14.

■

6.2-2 *Starting with the procedure* MAX-HEAPIFY, *write pseudocode for the procedure* MIN-HEAPIFY(A, i) *which performs the corresponding manipulation on a min-heap. How does the running time of* MIN-HEAPIFY *compare to that of* MAX-HEAPIFY *?*

Solution :

Algorithme 12. MIN-HEAPIFY (A, i)

```
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] < A[i]
4         least = l
5   else least = i
6   if r ≤ A.heap-size and A[r] < A[least]
7         least = r
8   if least ≠ i
9         swap(A, i, least)
10        MIN-HEAPIFY(A, least)
```

L'algorithme étant la même que celle du Max-Heapify à quelques signes près, on est certain qu'ils ont la même complexité en temps. Détaillons quand même la preuve du livre. Le temps d'exécution de Min-Heapify est le temps $\Theta(1)$ nécessaire pour corriger les relations entre les éléments, plus le temps d'exécuter Min-Heapify sur un sous-arbre enraciné sur l'un des enfants du nœud i.

Majorons la taille du sous-arbre de gauche. On sait que le pire des cas survient quand la dernière rangée de l'arbre est remplie à moitié, alors on a la ratio suivant :

$$R_n = \frac{\text{taille de l'arbre enraciné sur le nœud } \textsc{Left}(i)}{\text{taille de l'arbre enraciné sur le nœud } i} = \frac{2^{n-1}-1}{2^n - 2^{n-2} - 1}.$$

R_n est croissante et admet une limite $R_n \underset{n\to+\infty}{\longrightarrow} 2/3$ qui est le majoré de la taille du sous-arbre de gauche. Ainsi, on obtient l'inégalité :

$$T(n) \leq T(2n/3) + \Theta(1).$$

Puis on utilise le *master theorem*. Dans notre cas $f(n) = \Theta(1)$ et $n^{\log_{3/2} 1} = 1$. Le deuxième cas convient, on a bien $f(n) = \Theta(1)$, donc la solution est $T(n) = O(\lg n) = O(h)$ avec h l'hauteur de l'arbre (d'après 6.1-2, $h = \lfloor \lg n \rfloor$). ■

6.2-3 *What is the effect of calling* Max-Heapify(A, i) *when the element* $A[i]$ *is larger than its children ?*

Solution : L'algorithme effectue les comparaisons nécessaires et s'arrête en temps constant. Aucun échange a lieu car le nœud et ses fils vérifient la propriété d'un *max-heap*. ■

6.2-4 *What is the effect of calling* Max-Heapify(A, i) *for* $i >$ A.*heap-size*$/2$ *?*

Solution : On a $\lfloor$A.*heap-size*$/2\rfloor + 1 \leq i \leq$ A.*heap-size*, d'après 6.1-7, tous les nœud d'indice i sont des feuilles. Donc l'algorithme s'arrête en temps constant et aucun échange a lieu. ■

6.2-5 *The code for* Max-Heapify *is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient* Max-Heapify *that uses an iterative control construct (a loop) instead of recursion.*

Solution :

Algorithme 13. ITERATIVE-MAX-HEAPIFY (A, i)

```
1  while LEFT(i) < A.heap-size and RIGHT(i) < A.heap-size
2      l = LEFT(i)
3      r = RIGHT(i)
4      // Les lignes 5-9 peuvent être exprimer autrement [a]
5      if A[l] > A[i]
6          largest = l
7      else largest = i
8      if A[r] > A[largest]
9          largest = r
10     if largest ≠ i
11         swap(A, i, largest)
12         i = largest
13     else return
```

a. *largest* = index-of-max(A[i], A[l], A[r]).

■

6.2-6 *Show that the worst-case running time of* MAX-HEAPIFY *on a heap of size n is $\Omega(\lg n)$. (Hint : For a heap with n nodes, give node values that cause* MAX-HEAPIFY *to be called recursively at every node on a simple path from the root down to a leaf.)*

Solution : Le pire cas a lieu lorsqu'il existe un chemin tel que ses éléments sont triés de manière croissante. Dans ce cas là, il y a exactement $h = \lfloor \lg n \rfloor$ appels recursives de la fonction MAX-HEAPIFY. D'où la borne inférieure $\Omega(\lg n)$. ■

6.3 Building a heap

6.3-1 *Using Figure 6.3 as a model, illustrate the operation of* BUILD-MAX-HEAP *on the array* $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

Solution : Observons la figure (3) : (a) Un tableau de 9 éléments en entrée, avec l'arbre binaire qu'il représente. La figure montre que l'indice de boucle i pointe vers le nœud 4 avant l'appel MAX-HEAPIFY(A, i). (b) La structure de données résultante. L'indicde de boucle i de l'itération suivante pointe vers le nœud 3. (c)-(d) Les itérations suivantes de la boucle **for** de BUILD-MAX-HEAP. À chaque fois qu'il y a appel de MAX-HEAPIFY sur un nœud, les deux sous-arbres de ce nœud sont des *max-heap*. (e) Le *max-heap* après que BUILD-MAX-HEAP a fini.

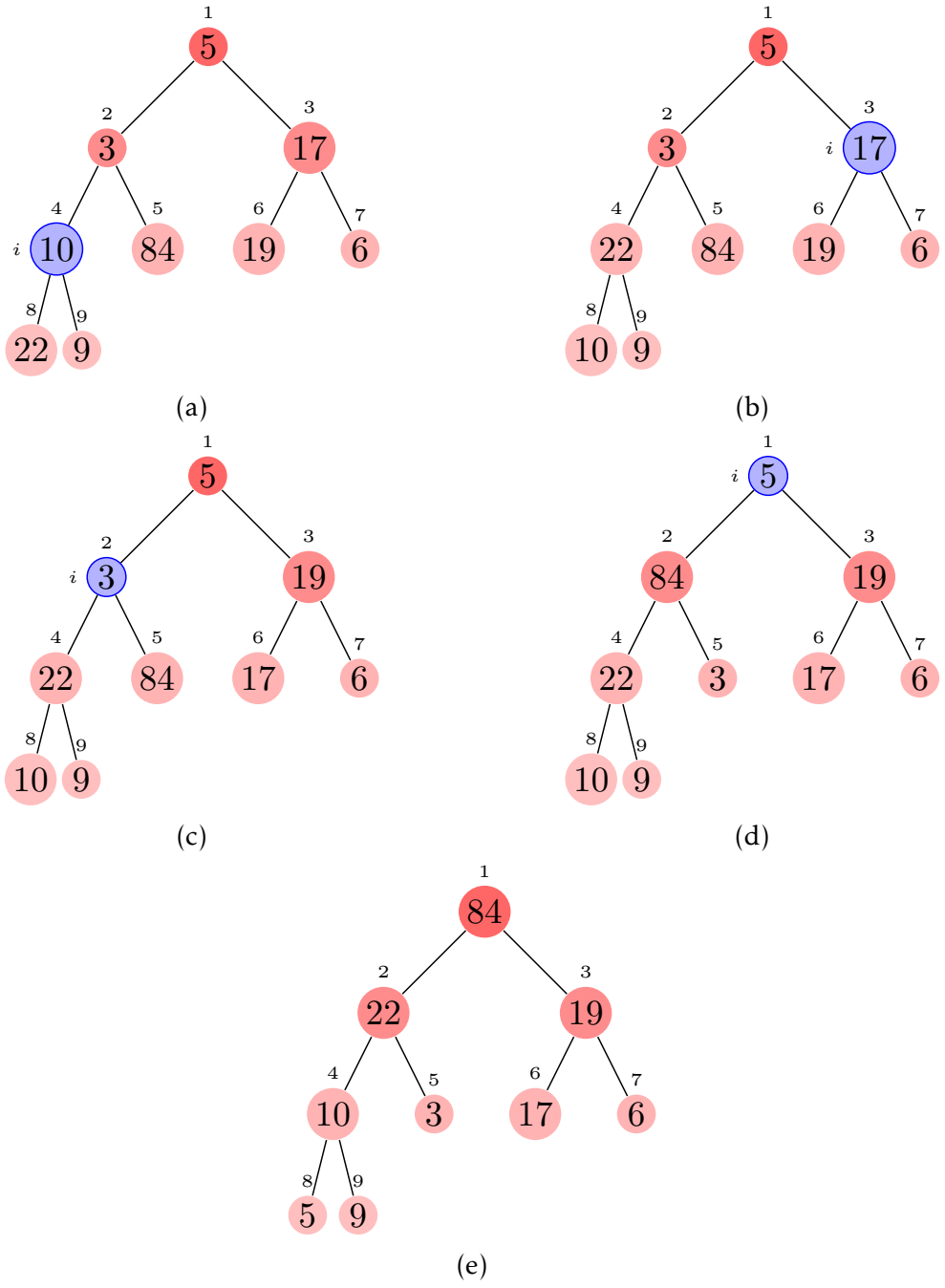

FIGURE 3 – Le fonctionnement de BUILD-MAX-HEAP(A), où A.*heap-size* = 9.

■

6.3-2 *Why do we want the loop index i in line 2 of* BUILD-MAX-HEAP *to decrease from* ⌊A.*length*/2⌋ *to* 1 *rather than increase from* 1 *to* ⌊A.*length*/2⌋*?*

Solution : L'hypothèse de MAX-HEAPIFY(A, i) est que les sous-arbres LEFT(i) et RIGHT(i) sont des *max-heaps*. Si l'on commence par le nœud 1, alors il n'y a aucune garantie que les nœuds parcourus seront toujours des *max-heaps*. ■

6.3-3 *Show that there are at most* $\lceil n/2^{h+1} \rceil$ *nodes of height* h *in any* n*-element heap.*

Solution : Notons que les nœuds ayant une hauteur j deviennent des feuilles après qu'on ôte ceux qui ont comme hauteur $0,\dots,j-1$. Sachant qu'un tas ayant n éléments a exactement $n-\lfloor n/2\rfloor = \lceil n/2\rceil$ feuilles (d'après l'exercice 6.1-7 et une identité de partie entière du chapitre 3), on peut établir deux suites récurrentes :

$$\begin{cases} N_0 = n \\ N_{i+1} = N_i - F_i \end{cases} \qquad \begin{cases} F_0 = \lceil N_0/2\rceil \\ F_i = \lceil N_i/2\rceil \end{cases}$$

où N_i désigne le nombre de nœuds après avoir ôter les nœuds de hauteur réspectives $0,\dots,i-1$ et F_i le nombre de feuilles de ce nouveau arbre. À partir de ces relations, il en découle que

$$N_{i+1} = N_i - \lceil N_i/2\rceil = \lfloor N_i/2\rfloor \leq N_i/2$$

puis, par itération, on a $N_i \leq n/2^i$. On conclut que $F_i = \lceil N_i/2\rceil \leq \lceil n/2^{i+1}\rceil$. ■

6.4 The heapsort algorithm

6.4-1 *Using Figure 6.4 as a model, illustrate the operation of* Heapsort *on the array* $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4\rangle$.

Solution : Regardons la figure (4). (a) L'arbre binaire avant l'appel de Build-Max-Heap. (b) la structure de *max-heap* juste après sa construction par Build-Max-Heap. (c)-(j) Le *max-heap* juste après chaque appel de Max-Heapify. La valeur de i à ce moment est montrée. Seul les nœuds rouges restent dans le tas.

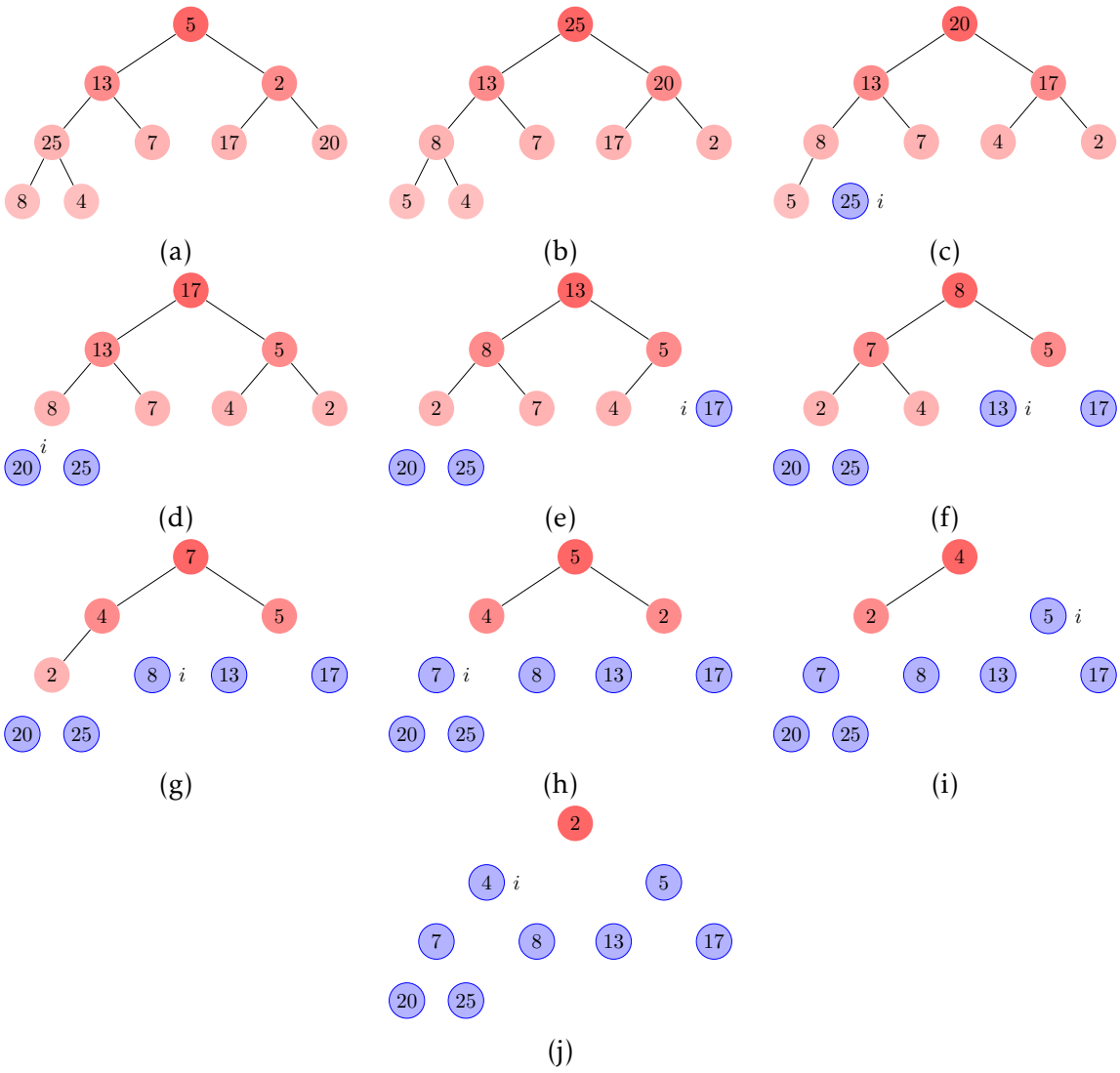

FIGURE 4 – Le fonctionnement du HEAP-SORT(A) sur $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

■

6.4-2 *Argue the correctness of* HEAP-SORT *using the following loop invariant :*
At the start of each iteration of the ***for*** *loop of lines 2–5, the subarray* $A[1..i]$ *is a max-heap containing the* i *smallest elements of* $A[1..n]$*, and the subarray* $A[i+1..n]$ *contains the* $n-i$ *largest elements of* $A[1..n]$*, sorted.*

Solution :

Initialisation : Avant la première itération de la boucle, $i = n$. De plus, par construction de BUILD-MAX-HEAP, $A[i..n]$ est un *max-heap*. Il contient le n-ième plus petit (autrement dit le plus grand) élément qui n'est rien d'autre que la racine du tas. Le sous-tableau $A[i+1..n]$ est vide, donc ne contient aucun élément trié.

Maintenance : Supposons, au début de l'itération à laquelle $i = k$, que le tableau $A[1..i]$ est un *max-heap* contenant le i-ième plus petit élément et que le tableau $A[i+1..n]$ contient les $n-i$ plus grands éléments de $A[1..n]$ triés d'ordre croissant. $A[1..i]$ étant un *max-heap*, $A[1]$ est donc le plus grand élément. En l'échangeant

avec $A[i]$, $A[i..n]$ contient les $n-i+1$ plus grands éléments de $A[1..n]$. En outre, en cas d'existence, LEFT(1) et RIGHT(1) sont des *max-heaps*, ainsi en appliquant MAX-HEAPIFY au nœud 1, il est garantie que $A[1..i-1]$ est un *max-heap*.

Terminaison : Après l'itération $i = 2$, $A[1]$ est le plus petit élément de $A[1..n]$ et $A[2..n]$ est trié de manière croissante, donc $A[1..n]$ est trié.

■

6.4-3 *What is the running time of* HEAP-SORT *on an array* A *of length n that is already sorted in increasing order? What about decreasing order?*

Solution :

■

6.4-4 *Show that the worst-case running time of* HEAPSORT *is* $\Omega(n \lg n)$.

Solution :

■

6.4-5 ⋆ *Show that when all elements are distinct, the best-case running time of* HEAP-SORT *is* $\Omega(n \lg n)$.

Solution :

■

6.5 Priority queues

6.5-1 *Illustrate the operation of* HEAP-EXTRACT-MAX *on the heap*
$A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

Solution : Considérons la figure (5). (a) Le tas A avant l'extraction. (b) L'extraction du plus grand élément $A[1] = 15$. (c)-(e) L'application de MAX-HEAPIFY$(A, 1)$. (f) On obtient à nouveau un *max-heap* et on retourne *max*.

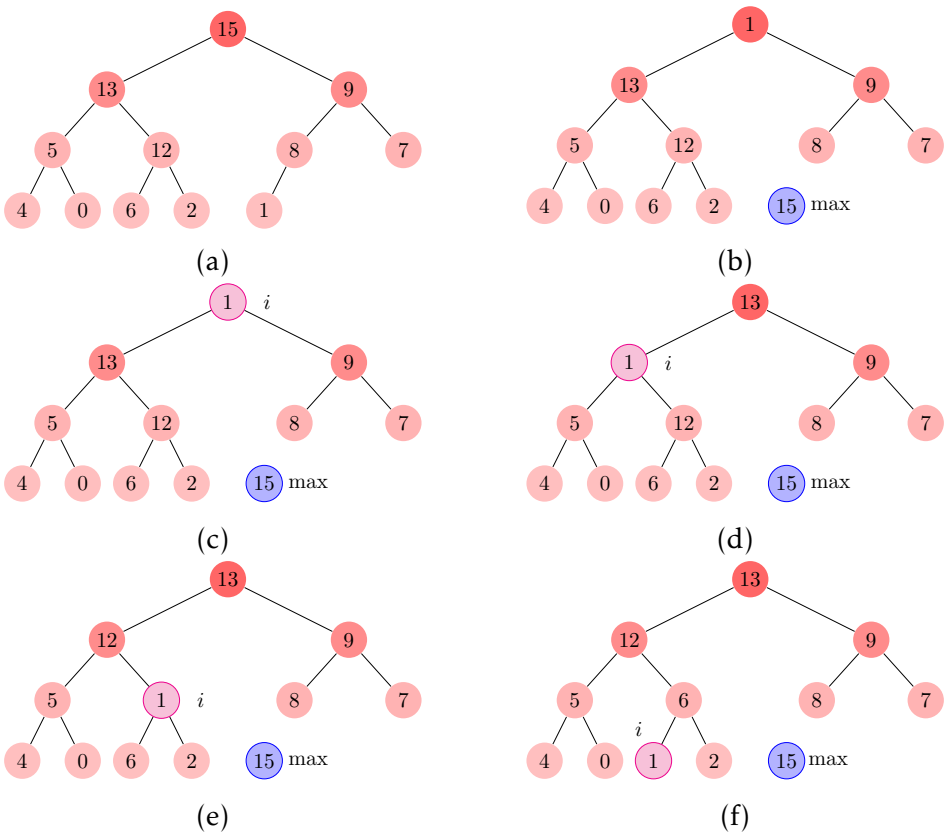

FIGURE 5 – Le fonctionnement de HEAP-EXTRACT-MAX(A) sur $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

■

6.5-2 *Illustrate the operation of* MAX-HEAP-INSERT *on the heap* $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

Solution :

■

6.5-3 *Write pseudocode for the procedures* HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, *and* MIN-HEAP-INSERT *that implement a min-priority queue with a min-heap.*

Solution :

■

6.5-4 *Why do we bother setting the key of the inserted node to* $-\infty$ *in line 2 of* MAX-HEAP-INSERT *when the next thing we do is increase its key to the desired value?*

Solution :

■

6.5-5 *Argue the correctness of* HEAP-INCREASE-KEY *using the following loop invariant :*

At the start of each iteration of the ***while*** *loop of lines 4–6, the subarray* $A[1..A.heap\text{-}size]$ *satisfies the max-heap property, except that there may be one violation : only* $A[i]$ *may be larger than* $A[\textsc{Parent}(i)]$.

You may assume that the subarray $A[1..A.heap\text{-}size]$ *satisfies the max-heap property at the time* Heap-Increase-Key *is called.*

Solution :

■

6.5-6 *Each exchange operation on line 5 of* Heap-Increase-Key *typically requires three assignments. Show how to use the idea of the inner loop of* Insertion-Sort *to reduce the three assignments down to just one assignment.*

Solution :

■

6.5-7 *Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.)*

Solution :

■

6.5-8 *The operation* $\textsc{Heap-Delete}(A, i)$ *deletes the item in node* i *from heap* A. *Give an implementation of* Heap-Delete *that runs in* $O(\lg n)$ *time for an* n*-element max-heap.*

Solution :

■

6.5-9 *Give an* $O(n \lg k)$*-time algorithm to merge* k *sorted lists into one sorted list, where* n *is the total number of elements in all the input lists. (Hint : Use a min-heap for* k*-way merging.)*

Solution :

■

7 Quicksort

7.1 Description of quicksort

7.2 Performance of quicksort

7.3 A randomized version of quicksort

7.4 Analysis of quicksort

8 Sorting in Linear Time

8.1 Lower bounds for sorting

8.2 Counting sort

8.3 Radix sort

8.4 Bucket sort

Deuxième partie

Data Structures

10 Elementary Data Structures

11 Hash Tables

12 Binary Search Trees

13 Red-Black Trees

14 Augmenting Data Structures

15 Dynamic Programming

16 Greedy Algorithms

Troisième partie

Advanced Data Structures

18 B-Trees

19 Fibonacci Heaps

20 van Emde Boas Trees

21 Data Structures for Disjoint Sets

Quatrième partie

Graph Algorithms

22 Elementary Graph Algorithms

22.1 Representations of graphs

22.1-1 *Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?*

Solution : Soient $G = (V, E)$ un graphe non-orienté, $v \in V$ un de ses sommets et $G.Adj$ sa représentation en liste adjacente. On peut calculer le degré sortant de v en calculant simplement la longueur de $G.Adj[v]$. Ainsi, le calcul de degré sortant de tous les sommets se fait en $\Theta(V + E)$.

Pour calculer le degré entrant de v on doit parcourir $G.Adj$ entièrement. De cette manière, il nous faudra $\Theta(VE)$ en temps. On peut aussi implémenter une liste de taille $|V|$, indexé par ses sommets dans lequel chaque élément est initialisé par 0. Dans ce cas là, il suffit de parcourir $G.Adj$ une fois en comptant chaque occurrence des sommets. Cette méthode a $\Theta(V + E)$ en complexité de temps et $\Theta(V)$ en espace. ■

22.1-2 *Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.*

Solution :

$$
\begin{array}{ll}
1: & 2 \quad 3 \\
2: & 4 \quad 5 \\
3: & 6 \quad 7 \\
4: & 2 \\
5: & 2 \\
6: & 3 \\
7: & 3
\end{array}
$$

(a) Liste adjacente

$$
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

(b) Matrice adjacente

FIGURE 6 – Représentation de l'arbre complet de 7 sommets en liste adjacente et matrice adjacente

■

22.1-3 *The* ***transpose*** *of a directed graph* $G = (V, E)$ *is the graph* $G^T = (V, E^T)$, *where* $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. *Thus,* G^T *is* G *with all its edges reversed. Describe efficient algorithms for computing* G^T *from* G, *for both the adjacency-list and adjacency-matrix representations of* G. *Analyze the running times of your algorithms.*

Solution : Pour transposer une liste adjacente, il est nécessaire de la parcourir entièrement, donc la complexité est de $\Theta(V + E)$.

Algorithme 14. TRANSPOSE-ADJ-LIST (*Adj*)

```
1  Create a new adjacency-list newAdj
2  for each linked-list i of Adj
3      for each element j of Adj[i]
4          newAdj[j].append(i)
```

Quant à une matrice adjacente, il suffit de parcourir les éléments supérieurs et les échanger à leur élément symétrique. De cette manière, il y a au total $\frac{n(n-1)}{2}$ d'échanges, d'où la complexité $\Theta(V^2)$.

Algorithme 15. TRANSPOSE-ADJ-MATRIX (M)

```
1  for i = 1 to M.size
2      for j = i + 1 to M.size
3          swap(M[i, j], M[j, i])
```

■

22.1-4 *Given an adjacency-list representation of a multigraph* $G = (V, E)$*, describe an* $O(V + E)$*-time algorithm to compute the adjacency-list representation of the « equivalent » undirected graph* $G' = (V, E')$*, where* E' *consists of the edges in* E *with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.*

Solution :

Algorithme 16. MULTIGRAPH-TO-UNDIGRAPH (*Adj*)

```
 1  Create an array A having length |V|
 2  for each linked-list Adj[i]
 3      init(A) // initialize all element to FALSE
 4      for each element j of Adj[i]
 5          e = Adj[i][j]
 6          if e == i // remove loop
 7              Adj[i].remove(j)
 8          else
 9              if A[e] == TRUE // remove multiple edge
10                  Adj[i].remove(j)
11              else
12                  A[e] = TRUE
```

■

22.1-5 *The* ***square*** *of a directed graph* $G = (V, E)$ *is the graph* $G^2 = (V, E^2)$ *such that* $(u, v) \in E^2$ *if and only if* G *contains a path with at most two edges between* u *and* v*. Describe efficient algorithms for computing* G^2 *from* G *for both the adjacency-list and adjacency-matrix representations of* G*. Analyze the running times of your algorithms.*

Solution : Dans le cas de matrice adjacente, on peut savoir l'existence d'une arrête en temps constant. Pour vérifier qu'il existe une arrête $(u, w) \in G^2$, on cherche toute paire d'arrête (u, v) avec v le sommet intermédiaire, puis en cas d'existence de $(u, v_0) \in G$, on cherche toute paire d'arrête $(v_0, w) \in G$. D'une telle manière, l'algorithme prend $\Theta(E^3)$.

Algorithme 17. SQUARE-OF-ADJ-MATRIX (M)

```
1  Create a matrix M_2 which have the same dimension of M
2  for each vertex i in M
3      for each vertex j in M[i]
4          if M[i][j] == 1
5              for each element k in M[j]
6                  if M[i][k] == 1
7                      M_2[i][k] = 1
8                  M_2[i][k] = 1
```

Algorithme 18. SQUARE-OF-ADJ-LIST (*Adj*)

```
1  Create a new adjacency-list newAdj
2  for each linked-list i of Adj
3      for each element j of Adj[i]
4          newAdj[i].append(j) // Add 1-length path
5          for each element k of Adj[j]
6              newAdj[i].append(k) // Add 2-length path
```

■

22.1-6 *Most graph algorithms that take an adjacency-matrix representation as input require time* $\Omega(V^2)$*, but there are some exceptions. Show how to determine whether a directed graph* G *contains a* ***universal sink*** *– a vertex with in-degree* $|V| - 1$ *and out-degree* 0 *– in time* $O(V)$*, given an adjacency matrix for* G*.*

Solution : On cherche, dans la matrice adjacente, une colonne qui contient $|V| - 1|$ 1. De cette manière, il suffit de parcourir qu'une fois le graphe entier, d'où la complexité $O(V)$.

Algorithme 19. SQUARE-OF-ADJ-LIST (*M*)

```
1  for each vertex in
```

■

22.1-7 *The* ***incidence matrix*** *of a directed graph* $G = (V, E)$ *with no self-loops is a* $|V| \times |E|$ *matrix* $B = b_{ij}$ *such that*

$$b_{ij} = \begin{cases} -1 & \text{if edge j leaves vertex i,} \\ 1 & \text{if edge j enters vertex i,} \\ 0 & \text{otherwise.} \end{cases}$$

Describe what the entries of the matrix product BB^T represent, where B^T is the transpose of B.

22.1-8 *Suppose that instead of a linked list, each array entry Adj[u] is a hash table containing the vertices v for which* $(u, v) \in E$. *If all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph? What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared to the hash table?*

Solution :

■

22.2 Breadth-first search

23 Minimum Spanning Trees

24 Single-Source Shortest Paths

25 All-Pairs Shortest Paths

26 Maximum Flow

27 Multithreaded Algorithms

28 Matrix Operations

29 Linear Programming

30 Polynomials and the FFT

31 Number-Theoretic Algorithms

32 String Matching

33 Computational Geometry

34 NP-Completeness

35 Approximation Algorithms

Cinquième partie

Appendix : Mathematical Background

A Summation

Références : [1]

A.1 Summation formulas and propreties

A.1-1 *Find a simple formula for* $\sum_{k=1}^{n}(2k-1)$

Solution :

$$\begin{aligned}\sum_{k=1}^{n}(2k-1) &= n(n+1)-n \\ &= n^2.\end{aligned}$$

■

A.1-2 ⋆ *Show that* $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + \mathcal{O}(1)$ *by manipulating the harmonic series.*

Solution :

$$\begin{aligned}\sum_{k=1}^{n}\frac{1}{2k-1} &= \mathrm{H}_n - \frac{1}{2}\sum_{k=1}^{n}\frac{1}{k} \\ &= \frac{1}{2}\mathrm{H}_n \\ &= \ln(\sqrt{n}) + \mathcal{O}(1).\end{aligned}$$

■

A.1-3 *Show that* $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$ *for* $0 < |x| < 1$.

Solution :

$$\begin{aligned}\sum_{k=0}^{\infty} k^2 x^k &= x\left(\frac{x}{(1-x)^2}\right)' \\ &= \frac{x}{(1-x)^2} + \frac{2x}{(1-x)^3} \\ &= \frac{x(1+x)}{(1-x)^3}.\end{aligned}$$

■

A.1-4 ⋆ *Show that* $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

Solution : Soit $S = \sum_{k=0}^{\infty} \frac{k-1}{2^k}$. Alors $2S = \sum_{k=0}^{\infty} \frac{k-1}{2^{k-1}} = -2 + \sum_{k=0}^{\infty} \frac{k}{2^k}$. Donc

$$\begin{aligned} 2S - S &= -2 + \sum_{k=0}^{\infty} \frac{1}{2^k} \\ &= 0. \end{aligned}$$

■

A.1-5 ⋆ *Evaluate the sum* $\sum_{k=1}^{\infty}(2k+1)x^{2k}$ *for* $|x| < 1$.

Solution :

$$\begin{aligned} \sum_{k=1}^{\infty}(2k+1)x^{2k} &= \left(\sum_{k=1}^{\infty} x^{2k+1}\right)' \\ &= \left(\frac{x}{1-x^2} - x\right)' \\ &= \frac{3x^2}{1-x^2} + \frac{2x^4}{(1-x^2)^2} \end{aligned}$$

■

A.1-6 *Prove that* $\sum_{k=1}^{n} \mathcal{O}(f_k(i)) = \mathrm{O}\left(\sum_{k=1}^{n} f_k(i)\right)$ *by using the linearity proprety of summations.*

Solution : Soit pour tout $k \in [\![1, n]\!], g_k \in \mathcal{O}(f_k)$, autrement dit $g_k \leq c_k f_k$ est vérifié à partir d'un rang N_k avec $c_k > 0$. On a donc $\sum_{k=1}^{n} g_k \leq \sum_{k=1}^{n} c f_k$ vérifié à partir d'un rang $N = \max(N_1, \ldots, N_n)$ et $c = \max(c_1, \ldots, c_n)$. D'où $\sum_{k=1}^{n} g_k \in \mathcal{O}(\sum_{k=1}^{n} f_k)$. ■

A.1-7 *Evaluate the product* $\prod_{k=1}^{n} 2 \cdot 4^k$.

Solution : Soit $P = \prod_{k=1}^{n} 2 \cdot 4^k = \prod_{k=1}^{n} 2^{2k+1}$. On a

$$\begin{aligned} \lg P &= \sum_{k=1}^{n} 2k+1 \\ &= n(n+2) \end{aligned}$$

Finalement, $P = 2^{n(n+2)}$. ■

A.1-8 ⋆ *Evaluate the product* $\prod_{k=2}^{n}(1 - 1/k^2)$.

Solution :

$$\begin{aligned}\prod_{k=2}^{n}(1-\frac{1}{k^2}) &= \prod_{k=2}^{n}\frac{k-1}{k}\cdot\frac{k+1}{k}\\ &= \frac{1}{2}\cdot\frac{n+1}{n}\\ &= \frac{n+1}{2n}.\end{aligned}$$

■

A.2 Bounding summations

A.2-1 *Show that $\sum_{k=1}^{n} 1/k^2$ is bounded above by a constant.*

Solution :

$$\begin{aligned}\sum_{k=1}^{n}\frac{1}{k^2} &= 1+\sum_{k=2}^{n}\frac{1}{k^2}\\ &\le 1+\int_1^n \frac{1}{x^2}\,\mathrm{d}x\\ &= 1+(1-\frac{1}{n})\\ &= 2-\frac{1}{n}\\ &\le 2.\end{aligned}$$

■

A.2-2 *Find an asymptotic upper bound on the summation $\sum_{k=0}^{\lfloor \lg n\rfloor}\lceil n/2^k\rceil$.*

A.2-3 *Show that the nth harmonic number is $\Omega(\lg n)$ by splitting the summation.*

A.2-4 *Approximate $\sum_{k=1}^{n} k^3$ with an integral.*

Solution : On a

$$\int_0^n x^3\,\mathrm{d}x \le \sum_{k=1}^{n} k^3 \le \int_1^{n+1} x^3\,\mathrm{d}x$$

ce qui donne

$$\frac{n^4}{4} \le \sum_{k=1}^{n} k^3 \le \frac{(n+1)^4-1}{4}.$$

Ainsi, $\sum_{k=1}^{n} k^3 = \Theta(n^4)$. ■

A.2-5 *Why didn't we use the integral approximation (A.12) directly on $\sum_{k=1}^{n} 1/k$ to obtain an upper bound on the nth harmonic number?*

Solution : Car la primitive de $1/x$ n'est pas définie en 0. ■

A.3 Problems

A-1 *Bounding summations*

Give asymptotically tight bounds on the following summations. Assume that $r > 0$ and $s > 0$ are constants.

a. $\sum_{k=1}^{n} k^r$.

b. $\sum_{k=1}^{n} \lg^s k$.

c. $\sum_{k=1}^{n} k^r \lg^s k$

Solution : ■

B Sets, Etc.

C Counting and Probability

C.1 Counting

C.1-1 *How many k-substrings does an n-string have? (Consider identical k-substrings at different positions to be different.) How many substrings does an n-string have in total?*

Solution : Il y a $S_k = n - k + 1$ k-sous-chaînes dans une n-chaîne. Le nombre total de sous-chaînes dans une n-chaîne est donc

$$\sum_{k=1}^{n} S_k = \frac{n(n+1)}{2}.$$

■

C.1-2 *An n-input, m-output boolean function is a function from $\{TRUE, FALSE\}^n$ to $\{TRUE, FALSE\}^m$. How many n-input, 1-output boolean functions are there? How many n-input, m-output boolean functions are there?*

Solution : Le cardinal de l'ensemble de fonction de E dans F est $|F|^{|E|}$. Particulièrement pour les fonctions booléenes, on a :

— pour $E = \{\text{TRUE}, \text{FALSE}\}^n$ et $F = \{\text{TRUE}, \text{FALSE}\}$, $|F|^{|E|} = 2^{2^n}$;

— pour $E = \{\text{TRUE}, \text{FALSE}\}^n$ et $F = \{\text{TRUE}, \text{FALSE}\}^m$, $|F|^{|E|} = (2^m)^{2^n}$.

■

C.1-3 *In how many ways can n professors sit around a circular conference table? Consider two seatings to be the same if one can be rotated to form the other.*

Solution : On considère tout d'abord le cas de sièges rangés linéairement. Dans ce cas, on a $n!$ arrangements. Dans le cas circulaire, on peut définir une relation d'équivalence $\mathcal{R}$ qui regroupe les arrangements qui peuvent être obtenus par une permutation circulaire. De ce fait, il y a exactement n éléments dans chaque classe. Le nombre d'arrangement de sièges placés circulairement est en fait le cardinal de l'espace quotienté par $\mathcal{R}$. Donc il y a $\frac{n!}{n} = (n-1)!$ arrangements. ■

C.1-4 *In how many ways can we choose three distinct numbers from the set $\{1, 2, \ldots, 99\}$ so that their sum is even?*

Solution : Pour obtenir un nombre pair en sommant trois nombres distincts de l'ensemble $E = \{1, 2, \ldots, 99\}$, il y a deux possibilités :

— pair + pair + pair = pair ;

— impair + impair + pair = pair.

Dans l'ensemble E, il y a 50 nombres impairs et 49 nombres pairs. Finalement, le nombre de combinaisons est :

$$\binom{49}{3}+\binom{50}{2}\binom{49}{1}=78449.$$

■

C.1-5 *Prove the identity*

$$\binom{n}{k}=\frac{n}{k}\binom{n-1}{k-1}$$

for $0<k\leq n$.

Solution :

$$\binom{n}{k}=\frac{n!}{k!(n-k)!}=\frac{n\cdot(n-1)!}{k(k-1)!((n-1)-(k-1))!}=\frac{n}{k}\binom{n-1}{k-1}.$$

■

C.1-6 *Prove the identity*

$$\binom{n}{k}=\frac{n}{n-k}\binom{n-1}{k}$$

for $0\leq k<n$.

Solution :

$$\binom{n}{k}=\frac{n}{n-k}\cdot\frac{(n-1)!}{k!((n-1)-k)!}=\binom{n-1}{k}.$$

■

C.1-7 *To choose k objects from n, you can make one of the objects distinguished and consider whether the distinguished object is chosen. Use this approach to prove that*

$$\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1}.$$

Solution :

On fixe un élément parmi les n et on en choisit k. Si l'élément est choisi, il y a $\binom{n-1}{k-1}$ moyens de choisir les autres ; sinon les k éléments choisis sont inclus dans les $n-1$ restants et il y a $\binom{n-1}{k}$ moyens de choisir. Conclusion : $\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1}$, d'où ce que montre la figure (7).

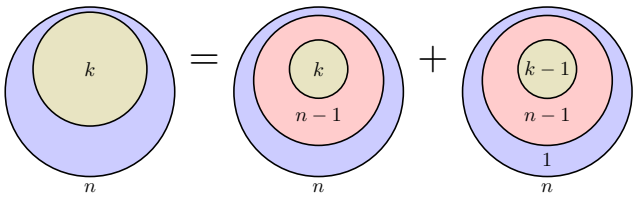

FIGURE 7 – « combinaison de k parmi n »

■

C.1-8 *Using the result of Exercise C.1-7, make a table for $n = 0, 1, \ldots, 6$ and $0 \leq k \leq n$ of the binomial coefficients $\binom{n}{k}$ with $\binom{0}{0}$ at the top, $\binom{1}{0}$ and $\binom{1}{1}$ on the next line, and so forth. Such a table of binomial coefficients is called* ***Pascal's triangle****.*

Solution :

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 \quad 1
\end{array}
$$

FIGURE 8 – Triangle de Pascal

■

C.1-9 *Prove that*

$$\sum_{i=1}^{n} i = \binom{n+1}{2}.$$

Solution :

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

■

C.1-10 *Show that for any integers $n \geq 0$ and $0 \leq k \leq n$, the expression $\binom{n}{k}$ achives its maximum value when $k = \lfloor n/2 \rfloor$ or $k = \lceil n/2 \rceil$.*

Solution : Il y a deux points de vue :

1. la symétrie : $\binom{n}{k} = \binom{n}{n-k}$;
2. la croissance : comparons $\binom{n}{k}$ et $\binom{n}{k+1}$:

$$\begin{aligned}
\binom{n}{k} < \binom{n}{k+1} &\iff \frac{n!}{k!(n-k)!} < \frac{n!}{(k+1)!(n-k-1)!} \\
&\iff k!(n-k)! > (k+1)!(n-k-1)! \\
&\iff (n-k)! > (k+1)(n-k-1)! \\
&\iff n-k > k+1 \\
&\iff n > 2k+1 \\
&\iff k < \frac{n-1}{2}.
\end{aligned}$$

En réunissant ces deux résultats, à n fixé, on observe que les coefficients $\binom{n}{k}$ pour $k = 0, \ldots, \lfloor \frac{n}{2} \rfloor$ croissent et atteignent une valeur maximale lorsque $k = \lfloor n/2 \rfloor$ et $k = \lceil n/2 \rceil$ (par symétrie). ■

C.1-11 ⋆

C.1-12 ⋆

C.1-13 ⋆

C.1-14 ⋆

C.1-15 ⋆

Références

[1] Jānis Lazovskis. University of Illinois at Chicago – MCS 401 Homework 1 grader solutions.

[2] Wikipedia. Convex hull — Wikipedia, The Free Encyclopedia, 2017.