Solutions du CLRS 3 *ed.*

Exercises : (41/21/957)
Problems : (0/6/158)

14 juin 2017

Table des matières

Liste des Algorithmes ii

1 The Role of Algorithms in Computing 1
- 1.1 Algorithms . . . 1
- 1.2 Algorithms as a technology . . . 1
- 1.3 Problems . . . 2

2 Getting Started 3
- 2.1 Insertion Sort . . . 3
- 2.2 Analyzing algorithms . . . 4
- 2.3 Designing algorithms . . . 6
- 2.4 Problems . . . 8

3 Growth of Functions 10
- 3.1 Asymptotic notation . . . 10
- 3.2 Standard notations and common functions . . . 10

4 Divide-and-Conquer 11
- 4.1 The maximum-subarray problem . . . 11
- 4.2 Strassen's algorithm for matrix multiplication . . . 12
- 4.3 The substitution method for solving recurrences . . . 15
- 4.4 The recursion-tree method for solving recurrences . . . 16
- 4.5 The master method for solving recurrences . . . 16

A Summation 19
- A.1 Summation formulas and propreties . . . 19
- A.2 Bounding summations . . . 21
- A.3 Problems . . . 22

B Sets, Etc. 23

C Counting and Probability 24
- C.1 Counting . . . 24

Références 26

Liste des Algorithmes

2.1.1 Decr-Insertion-Sort 3
2.1.2 Linear-Search 3
2.1.3 Add-Binary-Integer 4
2.2.4 Selection-Sort 5
2.3.5 Merge 6
2.3.6 Rec-Bin-Search 7
4.1.7 Brute-Find-Max-Subarray 11
4.1.8 Linear-Find-Max-Subarray 12
4.2.9 Add-Complex-Number 14

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 : ■

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

Solution : ■

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

Solution : ■

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

Solution : ■

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.*

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 : ■

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  n_1 = q − p + 1
2  n_2 = r − q
3  Let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4  for i = 1 to n_1
5      L[i] = A[p + i − 1]
6  for j = 1 to n_2
7      R[j] = A[q + j]
8  i = 1
9  j = 1
10 for k = p to r
11     if L[i] ≤ R[j] and i ≤ n_1
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* $1 \leq i \leq j+1$*. 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

```
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
```

■

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 :

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

— $n \times kn : \Theta(k n^{\lg 7})$

■

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.

■

1. voir Algorithme de Karatsuba (Wikipedia)

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* $\Theta(n \lg n)$.

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

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

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

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

Méthode de substitution : ■

4.5-5 ⋆ *Consider the regularity condition $af(n/b) \le cf(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of constant $a \ge 1$ 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. ■

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

C.1-5

C.1-6

C.1-7

C.1-8

C.1-9

C.1-10

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.