Algorithmique des Textes
Matthieu Boyer
Planche A

Khôlle MPI
30/09

Biggest Integer

Étant donné une liste d'entiers positifs, on va développer un algorithme les réarrangeant de sorte que les concaténer donne le nombre le plus grand possible.

1. Montrer qu'il est suffisant d'ordonner les entiers selon :

$$X \preceq Y \Leftrightarrow XY \leq YX$$

Pour cela, on va utiliser l'arbre des suffixes d'un mot. Celui-ci se construit en $\mathcal{O}(n)$ et est un trie pour le dictionnaire des suffixes. Un trie de D est un arbre dont chaque noeud est étiqueté par une lettre de sorte que :

— Pour chaque noeud les arêtes sortantes sont étiquetées par différentes lettres
— Pour chaque $S \in D$ il y a un chemin de la racine à un noeud qui épelle S. La fin du chemin est étiqueté par \$
— Chaque chemin de la racine à une feuille épelle un mot de D.

Étant donné un arbre des suffixes, on peut trouver en temps constant l'ancêtre commun le plus bas de deux noeuds.

2. Montrer qu'on peut utiliser l'arbre des suffixes pour comparer deux entiers en temps constant.
3. Donner une méthode pour trier tous les entiers avec au plus $\frac{\log_{10}(n)}{4}$ chiffres en $\mathcal{O}(n)$.
4. Montrer comment trier tous les entiers de taille plus que $\frac{\log_{10}(n)}{4}$ en $\mathcal{O}(n)$.
5. Donner l'algorithme résultat. Quelle est sa complexité en temps ? En espace ?

Solution: Biggest Integer

1. Since concatenation is associative, we can look at the concatenation of a list of length n as the concatenation of the numbers resulting of the concatenation of any partition $A_1, \dots, A_d$ of the list such that if $i < j$, $a_j \in A_k \Rightarrow a_i \in A_l$ where $l \leq k$. Thus, if the list is sorted, after concatenation of a part of its elements (provided they are neighbours), the obtained list remains sorted. Since for one and two elements, it is clear that results hold, we only need to cover the case of an insertion. Indeed, if sorting by insertion gives a correct result, then any sorting algorithm gives a correct result. Suppose that a list with n elements $a_0, \dots, a_{n-1}$ is sorted using the order defined. Let a_n be another number, and suppose it is inserted between a_i and a_{i+1} by insertion sort. Then, suppose that the result we obtain is not the right one. We return to the case of three elements, since the final concatenation can be seen as the one of the results of the concatenation of $(a_0, \dots, a_j)$, (a_n) and $(a_{j+1}, \dots, a_{n-1})$. Indeed, the values are still sorted using our order. The result obtained by the case of three elements would not be right, and thus the list would not be sorted. In the case of three elements, $a_0 \preceq a_1 \preceq a_2$. By checking all 6 permutations of the elements, we obtain the wanted result.
2. Consider two integers $X = X_1 \dots X_n$ and $Y = Y_1 \dots Y_m$. We suppose without loss of generality that $n \leq m$. We need to compare the strings $X_1 \dots X_n \mid Y_1 \dots Y_{m-n} \mid Y_{m-n+1} \dots Y_m$ and $Y_1 \dots Y_n \mid Y_{n+1} \dots Y_m \mid X_1 \dots X_n$. The vertical bars $\mid$ serve only as placeholders for the next step. We will first get the Lowest Common Ancestor (LCA) of X and Y. If it is strictly smaller than $|X|$, we return the comparison of the next digits in X and Y, else, if it is of length $|X|$, we go to next step : we apply the same operations to $Y_{n+1} \dots Y_m$ and Y. If we still have an equality, we then go to the final step and start over with $Y_{m-n+1} \dots Y_m$ and X.
3. There are $10^{\log_{10}(n)/4} = \sqrt[4]{n}$ numbers that, in base 10, are written using at most $\frac{\log_{10} n}{4}$ digits. Then, we can sort all the integers using counting sort : we create an array A of size $\sqrt[4]{n}$ with only zeros. We first need to sort the indices in this array using the relationship defined in 1), which is done using an optimal sorting algorithm in $\mathcal{O}(\sqrt[4]{n} \log n) = \mathcal{O}(n)$. Then, if we see i in the list of numbers with at most $\frac{\log_{10} n}{4}$ digits, we increase the slot in array corresponding to i by 1. This is done in $\mathcal{O}(n)$ since we see each digit at most once. Then, again, by iterating through $i \in [\![0, \sqrt[4]{n} - 1]\!]$ we can add i to a list $A[i]$ times. This is again done in $\mathcal{O}(n)$.
4. There are at most $\frac{n}{\frac{\log_{10} n}{4}}$ numbers with at least $\frac{\log_{10} n}{4}$ digits. Then using an optimal sorting algorithm, since the comparisons are done in $\mathcal{O}(1)$ by question 2), we get a complexity in :

$$\begin{aligned}
\mathcal{O}\left(\frac{4n}{\log_{10} n} \log_{10}\left(\frac{4n}{\log_{10} n}\right)\right) &= \mathcal{O}\left(4n \times \frac{\log_{10}(4) + \log_{10}(n) - \log_{10}(\log_{10}(n))}{\log_{10}(n)}\right) \\
&= \mathcal{O}\left(n \times \left(1 + \frac{\log_{10}(4)}{\log_{10}(n)} - \frac{\log_{10}(\log_{10}(n))}{\log_{10}(n)}\right)\right) \\
&= \mathcal{O}(n)
\end{aligned}$$

5. We will denote by $|X|$ the number of digits of X and by A, k the input array and its length.
   Based on the previous questions, we propose the following algorithm :
   (a) We compute the generalised suffix tree of the numbers. This is done in

$$\mathcal{O}\left(\sum_{i \in [\![0, k-1]\!]} |A[i]|\right) = \mathcal{O}(n)$$

   (b) We first sort the digits with at most $\frac{\log_{10} n}{4}$ digits. From this, by question 3), we can sort all the numbers is $\mathcal{O}(n)$ time.
   (c) Then, by question 4), we can sort the rest of the numbers in $\mathcal{O}(n)$ time.
   (d) We then merge the two sorted arrays in linear time in the sum of the lengths, thus in $\mathcal{O}(n)$.
   (e) Finally, we can compute the concatenation in $\mathcal{O}(n)$.

   This algorithm is correct by question 1), and has time complexity $\mathcal{O}(n)$.

Algorithmique des Textes
Matthieu Boyer
Planche B

Khôlle MPI
30/09

Codage d'Images

On définit une suite c_k de caractères binaires (ou bien N ou bien B) pour décrire une image noire et blanche. On s'intéresse pour $n \in \mathbb{N}$ à $S_{n^2} = c_0 \dots c_{n^2-1}$. On représentera une image par un tableau à deux dimensions de caractères. Dans une image de taille l par h, le pixel de coordonnées $(0,0)$ est en haut à gauche. On va encoder ces images en utilisant les positions des plus grands carrés blancs dans l'image. On définit un type image :

```
typedef struct image {
        char** pixels;
        uint largeur;
        uint hauteur;
} image;
```

On s'intéresse pour cela au problème suivant :

Plus Grand Carré Blanc	
Entrée:	Pixel (x, y) ; Matrice Image M
Sortie:	La taille du plus grand carré blanc de M dont le coin inférieur droit est (x, y). On pourra renvoyer 0 si le pixel est noir.

1. Proposer un codage d'image qui se base sur la recherche de plus grands carrés blancs. On ne cherchera pas à donner d'algorithme pour réaliser le codage pour l'instant. On rappelle que le taux de compression d'un codage est le complémentaire à 1 du rapport entre la mémoire occupée par l'image codée et l'image initialement. En donnant un exemple, donner le taux de compression optimal du codage, et son taux de compression dans le pire cas. On pourra s'intéresser à un échiquier.
2. Donner une relation de récurrence reliant les réponses à ce problème en (x, y) à celles des sous-problèmes en $(x-1, y), (x, y-1)$ et $(x-1, y-1)$.
3. Donner un algorithme de programmation dynamique pour trouver le plus grand carré d'une image. 0n spécificiera les structures de données utilisées. Donner la complexité spatiale de votre algorithme.
4. En vous inspirant de la solution par programmation dynamique du problème du plus grand carré blanc, proposer, étant donné un pixel (x, y), une heuristique gloutonne pour éliminer, après le calcul du plus sous-problème associé au pixel (x, y), un maximum de carrés blanc du codage sans perdre la couverture.
5. Implémenter l'algorithme de codage. Donner sa complexité en temps et en espace.
6. Implémenter l'algorithme de décodage.

Solution: Codage d'Images

1. On encore un carré blanc comme trois entiers (longueur du côté et coin bas droit). Le codage d'une image de dimension $l \times h$ sera donc une séquence de triplés d'entier $t_0 \cdot t_1 \cdot \ldots t_n$ de telle sorte que $t_1, \ldots, t_n$ soit les codages des n carrés blancs couvrant l'ensemble des pixels blancs de l'image et $t_0 = n \cdot l \cdot h$.
2. On a clairement $PGCB(x, y) = 1 + \min(PGCB(x-1, y), PGCB(x, y-1), PGCB(x-1, y-1))$ si (x, y) n'est pas noir, 0 sinon.
3. cf : `https://anthonylick.com/wp-content/uploads/correction_kholle6.pdf`

Tries, Aho-Corasick

Définition 0.1 Un dictionnaire D est un ensemble de string. Un trie de D est un arbre dont chaque noeud est étiqueté par une lettre de sorte que :

- Pour chaque noeud les arêtes sortantes sont étiquetées par différentes lettres
- Pour chaque $S \in D$ il y a un chemin de la racine à un noeud qui épelle S. La fin du chemin est étiqueté par \$
- Chaque chemin de la racine à une feuille épelle un mot de D.

1. Écrire le trie pour $\{abc, bc, ac, a\}$.
2. Montrer qu'un trie pour D utilise $\mathcal{O}\left(m = \sum_{S \in D} |S|\right)$ d'espace.

On définit le maillon faible d'un noeud u étiqueté par un string S comme une arête vers le noeud v le plus profond étiqueté par un suffixe propre de S.

3. Montrer que le trie demande $\mathcal{O}(m)$ espace et peut être construit en $\mathcal{O}(m)$.

On va essayer de résoudre le problème suivant :

Multiple Pattern Matching	
Entrée:	D un dictionnaire, T un texte
Sortie:	Les occurences de tous les mots de D dans T

.

4. On suppose d'abord qu'aucun mot de D n'est un suffixe d'un autre mot de D. Donner un algorithme se basant sur un Trie pour résoudre le problème. Vous n'aurez le droit qu'à $\mathcal{O}(m)$ espace et $\mathcal{O}(m + |T|)$ temps.
5. Modifier l'algorithme précédent en s'autorisant $\mathcal{O}(m + |T| + occ)$ temps pour répondre au problème.

Solution: Tries, Aho-Corasick

1. On étale les chemins.
2. On a :

Space : There are $\mathcal{O}(m)$ nodes, each node has exactly one failure link.

Time : Failure links are built top-to-down. Links from nodes of depth 1 point to the root. Suppose that we have built links for all nodes to depth $\leq d-1$. The failure link from a node labeled with Sa must point to a node labeled with $S'a$, where S' is a suffix of S. In other words, the parent of the end of the failure link from the node must belong to the failure link path from the node $p(u)$, and it must be the deepest node that has an outgoing edge labeled with a. The algorithm works as follows : we follow the failure link path from $p(u)$. The first node with an outgoing edge labeled with a is the end of the failure link for u. Consider the time needed to build the failure links for one root-to-leaf path. Let $root, u_1, \ldots, u_k$ be the nodes in this path, and denote by $f(u_i)$ the depth of the end of the failure link for u_i. The time to find the link for u_i is $\leq c \times (2 + f(u_{i-1} - f(u_i)))$. Thus the total time for the trie is $\mathcal{O}(m)$.

3. — Space : m is the total length of the patterns
   — Time : We need $\mathcal{O}(m)$ time to build the trie. Consider how the depth of *curr_node* changes during the algorithm.
     — Every time we do down an edge (happens $\leq n$ times), it increases by 1. Every time we follow a failure link, it decreases by ≥ 1.
     — Therefore, as the depth is always positive, we follow a failure link at most n times.

```
Input T, D, n
curr_node ← root, i ← 1, L ← []
while i ≤ n do
    if ∃e = (curr_node, u) labeled with T[i] then
        curr_node ← u
        if curr_node corresponds to a pattern then
            L ← i :: L
        i ← i + 1
    else
        if curr_node = root then
            i ← i + 1
        else
            curr_node = failure_link(curr_node)
```

4. When patterns are substrings of others, we add output links : an output link from a node u goes to the nearest node on the failure link path outgoing from u that corresponds to a pattern of the dictionary. Output links can be built in $\mathcal{O}(m)$ time by one top-down traversal of the failure link tree. We modify the algorithm : at each step it follows the path from *curr_node* and outputs the patterns in the output link path. The multiple pattern matching problem can ve solved in $\mathcal{O}(m)$ space and $\mathcal{O}(n + m + occ)$ where occ is the total number of occurences of the patterns in the text.