Homework Assignment 1

Matthieu Boyer

11 octobre 2023

Table des matières

1 Exercise 1 - [Edit Distance/Levenshtein Distance]

1.1 Question 1

Proposition 1.1.1 (Complexity and Correction)**.** *If we denote C_f the complexity of f, this algorithm has time complexity $\mathcal{O}\left(C_f\left(\frac{n}{t}\right)^2\right)$. This algorithm is correct.*

Démonstration. — Moreover, it is clear this algorithm is correct as it only just applies the dynamic programming algorithm for the Levenshtein distance by steps.

— This algorithm complexity comes from the fact it has two while loops for which the commands are executed at most n/t times. The commands in both *while* loops are executed in $\mathcal{O}(C_f)$. The *for* loops inside the *left while* loop are equivalent to loops for i between left and left + $t-1$ and thus are disjoint. The sum of their complexity over the *left while* loop is then n. The number of operations inside the *up while* loop is then in $\mathcal{O}\left(C_f\frac{n}{t}\right)$ and thus the total complexity is, as announced, in $\mathcal{O}\left(C_f\left(\frac{n}{t}\right)^2\right)$

∎

1.2 Question 2

By the recurrence formula : $\mathbf{D}[i][j] = \max\begin{cases}\mathbf{D}[i-1][j]+1\\ \mathbf{D}[i][j-1]+1\\ \mathbf{D}[i-1][j-1]+1 if S[i]\neq T[j] \text{ else } 0\end{cases}$ we see that $\mathbf{D}[i][j]$ is at most 1 plus one of its left neighbour, upper neighbour or upper left corner neighbour.

Algorithme 1 Question 1 - Levenshtein Distance with f

Input S, T, f, t ▷ Two Strings, the function f computing the values and the step t

$\mathbf{D} = \text{zeros}(n+1, n+1)$ ▷ $len(S) = len(T) = n$
for $i \leftarrow 0$to $n+1$ **do**
  $\mathbf{D}[i][0] \leftarrow i$
end for

for $j \leftarrow 0$ to $n+1$ **do**
  $\mathbf{D}[0][j] \leftarrow j$
end for

$\text{up}, \text{left} \leftarrow 0, 0$
while $\text{up} < n$ **do**
  $\text{left} \leftarrow 0$
  while $\text{left} < n$ **do**
    $down \leftarrow \min(n - \text{up}, t)$
    $right \leftarrow \min(n - \text{left}, t)$

    $b \leftarrow \mathbf{D}[\text{up}][\text{left}]$
    $a \leftarrow \mathbf{D}[\text{up}+1 \to \text{up}+1+down][\text{left}]$
    $c \leftarrow \mathbf{D}[\text{up}][\text{left}+1 \to \text{left}+1+right]$

    $f(a, b, c, d, e)$ ▷ We can suppose here that f modifies only the last line and column of F in $\mathbf{D}$ with side-effect.
    $\text{left} \leftarrow \text{left} + right$
    for $i \leftarrow 1$ to $down - 1$ **do**

$$\mathbf{D}[\text{up}+i][\text{left}] \leftarrow \min \begin{cases} \mathbf{D}[\text{up}+i][\text{left}-1]+1 \\ \mathbf{D}[\text{up}+i-1][\text{left}]+1 \\ \mathbf{D}[\text{up}+i-1][\text{left}-1]+\mathbb{1}_{\{S[up+i]=T[left]\}} \end{cases}$$

▷ We update the first Column of the block we consider.
    end for
  end while
  $\text{up} \leftarrow \text{up} + d$
  for $i \leftarrow 1$ to n **do**

$$\mathbf{D}[\text{up}][i] \leftarrow \min \begin{cases} \mathbf{D}[\text{up}][i-1]+1 \\ \mathbf{D}[\text{up}-1][+i]+1 \\ \mathbf{D}[\text{up}-1][i-1]+\mathbb{1}_{\{S[up+i]=T[left]\}} \end{cases}$$

▷ We update the first line of the blocks we will consider.
  end for
end while
return $\mathbf{D}[n][n]$

1.3 Question 3

By recurrence formula, if we substract from all the values in A, B, C a certain integer k, then we get for the new matrix F, the one we would have gotten with A, B, C with k substracted to all values. Thus, if the values in A, A', B, B' and C, C' all differ from a common integer, the resulting values after applying f will differ from this same value. Thus, $F' = F + (A^{'} - A)$ and $A^{'} - A$ is a matrix with all values equal.

1.4 Question 4

We will show here that we can pre-compute all $t \times t$ matrices in $\mathcal{O}\left(3^{2t}\sigma^{2t}t^2\right)$.
First, as $\mathbf{D}[i][j]$ here is the minimal number of elementary operations to go from string $S[0 : i]$ to string $T[0 : j]$. We can thus interpret the submatrix of $\mathbf{D}$ between (i, j) and $(i + t, j + t)$ as the dynamic programming matrix for minimum number of operations to go from string $S[i : i + t + 1]$ to string $T[j : j + t + 1]$ to which we added a first line and a first column.
We thus see that those matrices can be fully determined by their first line, first column and by two words.
As the values in $\mathbf{D}$ are bounded by 0 and $2n$ (we can always remove all letters in S and add all letters in T), and by question 2., as values along a line or a column differ by an integer in $-1, 0, 1$, the number of first lines and columns is $\mathcal{O}(n3^{2t})$. However, from question 3., if we allow negative values for f (which we have no reason not to do), we can always substract from the first line and column the value in the top left corner, and re-add it in $\mathcal{O}(t^2)$ after pre-processing.
Moreover, there are σ^t words of length t over the alphabet so the number of submatrices is $\mathcal{O}(3^{2t}\sigma^{2t})$.
We can then use the recurrence equation to derive the values on the submatrix in $\mathcal{O}(t^2)$.
Finally, we can pre-compute all $t \times t$ submatrices in $\mathcal{O}(3^{2t}\sigma^{2t}t^2)$.
Then, to access the values of the submatrix from $(up, left)$ to $(up{+}t, left{+}t)$, we need to identify the preprocessed corresponding matrix and thus we need to go through $S[up : up{+}t]$, $T[left : left{+}t]$, the first column and row of this submatrix to which we substracted the upper left value, which all are done in $\mathcal{O}(t)$.

1.5 Question 5

From Question 4, we have got an algorithm that allows us to compute the result in $\mathcal{O}(3^{2t}\sigma^{2t}t^2 + \left(\frac{n^2}{t}\right))$.
Indeed, after pre computing, we only need to check in $\mathcal{O}(1)$ the $\mathcal{O}\left(\left(\frac{n}{t}\right)^2\right)$ $t \times t$ submatrices in $\mathbf{D}$.
Thus, if we take $t = \log_{(3\sigma)}(\sqrt{n})$, we have complexity in $\mathcal{O}\left((3\sigma)^{\log_{3\sigma}(n)} \log_{3\sigma}(\sqrt{n}) + \left(\frac{n^2}{\log_{3\sigma}(\sqrt{n})}\right)\right)$.
As $\log_{3\sigma}(\sqrt{n}) = \Theta(\log(n))$ and $(3\sigma)^{\log_{3\sigma}(n)} \log_{3\sigma}(\sqrt{n}) = \mathcal{O}(n \log(n)^2) = o(\frac{n^2}{\log(n)})$, this algorithm has complexity in $\mathcal{O}\left(\frac{n^2}{\log n}\right)$

2 Exercice 2

In this exercise, we will denote by $\lg(n)$ the log in base 2 of n

2.1 Question 1

We will try to precompute arrays containing part of the answer by dividing the bitvector B into multiple blocks, of smaller and smaller size. We will use two intermediate arrays of size $\mathcal{O}\left(\frac{n}{\lg(n)}\right) = o(n)$ bits and a final array (meaning lowest level array) of size $\mathcal{O}(n)$.

We build a first array A_0 which will contain blocks of size $s_0 = \lg^2(n)$ (we will justify this value later). The blocks of this array contain the rank of the first position in B that is compacted in the block, i.e., $A[i] = \mathsf{rank}_1(i \times s_0)$ where rank designates the rank operation in B. As each entry in A takes at most $\lg(n)$ bits to store, A takes $\mathcal{O}\left(\frac{n}{\lg^2(n)}\lg(n)\right) = \mathcal{O}\left(\frac{n}{\lg(n)}\right) = o(n)$ bits to store, thus justifying the value chosen for s_0.

Then we build a second array A_1 which will store the ranks of smaller blocks of size $s_1 = \lg(n)/2$ (as for s_0, we will justify this value later). Here, the items in the array only need at most $\lg\left(\lg^2(n)\right)$ bits to be stored. This table thus takes $\mathcal{O}\left(\frac{n}{s_1}\lg\left(\lg^2(n)\right)\right) = \mathcal{O}\left(4n\frac{(\lg\lg(n))}{\lg(n)}\right) = o(n)$ bits to store, the equality coming from the properties of lg, thus explaining the general form of s_1 as $\lg * c$ but not yet why $c = 1/2$

To finally answer the query, we need to maintain a third array A_2 which will contain correspondances between every possible $\mathsf{rank}_1(i)$ query on a bitvector of length s_1 with $i < s_1$ and their answers. There are 2^{s_1} such bitvectors, and storing all answers on each can be done in $\mathcal{O}(s_1\lg(s_1))$. A_2 thus needs $\mathcal{O}(2^{s_1}\lg(s_1)s_1) = \mathcal{O}(\sqrt{n}\lg(n)\lg(\lg(n))) \leq n$, as $\lg(n)/2 = \lg(\sqrt{n})$ and from the properties of lg, thus explaining the coefficient of lg in s_1.

Using these arrays we can compute $\mathsf{rank}_1(i)$ in $\mathcal{O}(1)$ time by searching in blocks i/s_0, $(i \mod s_0)/s_1$ and $(i \mod s_0) \mod s_1$ in A_0, A_1 and A_2, and as : $\mathsf{rank}_0(i) = i - \mathsf{rank}_1(i)$, we obtain the wanted time complexity. The total number of bits used by these arrays is at most $n + o(n)$ so we have the wanted space complexity, and this solution is one.

2.2 Question 2

The solution proposed before can lead to a $\mathcal{O}(\lg(n))$ time complexity, so it is not sufficient to answer the select queries. We will thus refine the latter to get a new structure, by using three levels of intermediate arrays using $o(n)$ bits and two final arrays using $\mathcal{O}(n)$ bits.

We define our first array C_0 which records the positions of the $\lg^2(n)$-th (again, this value will be explained later) 1 bits. Storing a value in this array costs at most $\lg(n)$ bits and there are at most $\frac{n}{\lg^2(n)}$ values stored in the array. So it only uses : $\frac{n}{\lg(n)}$ bits.

Then, we will repeat the operation. Let s_1 be the size of a block in our first array. We want to use at most $\frac{s_1}{\lg(n)} \leq \lg(n)$ bits on this block in our second array C_1. We can calculate s_1 on the fly in $\mathcal{O}(1)$ during calculation, and using this, we can deduce the location in B of this block. However, there are $\lg^2(n)$ answers in that range, and it so requires $\lg(n)^3$ bits to store. Then, if we have $s_1 \geq \lg(n)^4$, we would have sufficient space to store the values in $o(n)$ bits, but, in the other case, we need to redivide the block in the same way.

We denote by s_2 the size of the considered block in C_1. We would need $\lg(s_2)\lg(s_1)\lg(n)$ bits to store all the answers. We obtain a similar inequality on s_2 as on s_1 before, by the same methods. If we have $s_2 \geq \lg(s_2)\lg(s_1)\lg(n)$ then we have sufficient space not to need to go further in the construction and keep $o(n)$ space. Else, we have :

$$s_2 < \lg(s_2)\lg(s_1)\lg(n)$$
$$s_1 < (\lg(n))^4$$

So we have : $\lg(s_1) < 4(\lg(\lg(n)))$. Then, by growing of lg and as $s_2 < s_1$ we get : $\lg(s_1) < 4\lg(\lg(n))$ and thus :

$$s_2 < 16\lg(n)\lg^2(\lg(n))$$

.

Using the same idea as in question 1., we answer select queries using final arrays, storing the values we need. Indeed, for each of the 2^{s_3} bit pattern of length $s_3 = \frac{\lg(n)}{2}$, we can record the position of the i-th 1 bit in the pattern in an array C_3, and store the number of 1 in the pattern in an array C_4. Then to compute the value of $\mathsf{select}_1(i)$ we can scan the range using C_4 to know which subrange contains the answer and use C_3 to get the answer, all in time $\mathcal{O}(1)$.

Finally, $\mathsf{select}_1(i)$ can be computed by finding the block in C_0 in which i is. From here, we compute s_1 and then, based on the case disjunction detailed earlier, we either get the correct answer from C_1 or compute s_2 and start over, using the final arrays. Thus, this data structure supports select queries in $\mathcal{O}(1)$ time. For storing the auxiliary directories C_0, C_1 and C_2, we need at most $\frac{3n}{\lg(n)}$ bits (the simplicity of this expression justifies the values chosen for s_1 and s_2), and for the final arrays we use $\sqrt{n}\left(\lg(n) + \frac{1}{2}\lg(n)\lg(n)\right)$, by the same calculation as in question 1, thus justifying the choice for the size of the patterns in C_4. The extra storage is in $\mathcal{O}(n)$, and this data structure is compatible with our requirements.

2.3 Question 3

Here, we model our set S of items as a bitvector B of length $n = \max S$ with coefficients :

$$B[i] = 1 \text{ if } i \in S \text{ else } 0$$

Then, solving, the predecessor problem can be done using the data structures introduced in questions 1 and 2. Indeed, finding the predecessor of j in S can be done by calling $\mathsf{rank}_1(j)$, getting the number of elements smaller than j in S, then calling select_1 on the resulting value, giving us the greatest element in S that is smaller than j.

2.4 Question 4

We first proceed by induction to prove that the depth of the tree is in $\mathcal{O}(\log_2(n))$

- Initialization : For all $c > 0$, the depth of the tree is $0 \leq c\log_2(1) = 0$.
- Heredity : Let P be a set of points of size n. We let P_0 and P_1 as in the definition. We have $|P_0| + |P_1| = |P| = n$. Let $a = |P_0|$. If $a = n$ or $a = 0$, there is nothing to inspect, since we can directly reduce the size of the rectangle we are studying. Else, by induction hypothesis, since both P_0, P_1 have size at most $n-1$, the left subtree has depth at most $c\log_2(a)$ and the right subtree has depth at most $c\log_2(n-a)$. Thus, we get the depth of the full tree to be : $c\max(\log_2(a), \log_2(n-a)) + 1 \leq c\log_2(n)$ if $c \geq 1$ by concavity of logarithm.

Thus, by induction, the depth of a wavelet tree is in $\mathcal{O}(\log_2(n))$.

Then, since we store in each depth bitvectors whose sum of lengths in n, along with the data structures from questions 1 and 2.
We so use $\mathcal{O}(\log_2(n) \times (n + o(n) + \mathcal{O}(n))) = \mathcal{O}(n\log_2(n))$

2.5 Question 5

The points in $\{(x, y) \in P_0 : x_1 \leq x \leq x_2\}$ are the points which are in position $i \in [x_1, x_2]$ in B with $B[i] = 1$. Thus we get that this interval corresponds in B_0 to the interval

$$[\mathsf{rank}_0(x_1 - 1) + 1, \mathsf{rank}_0(x_2)]$$

The same goes with P_1 by replacing the queries by their opposite value element. Note that this is done in $\mathcal{O}(1)$ from the first questions.

2.6 Question 6

We construct a wavelet tree over $[1, n]$ using the method defined earlier. Then we use the following to answer the query.

Algorithme 2 Orthogonal Range Query

```
function (ORQ)
    Input B, x_1, x_2, y_1, y_2, m, M
    if [m, M] ∩ [x_1, x_2] = ∅ then
        return 0
    end if
    if [m, M] ⊆ [x_1, x_2] then
        return x_2 − x_1 + 1                      ▷ These values will be maintained
    end if

    tmp ← ⌊(m+M)/2⌋
    [x_{1,0}, x_{2,0}] ← [rank_0(B, x_1 − 1) + 1, rank_0(B, x_2)]
    [x_{1,1}, x_{2,1}] ← [rank_1(B, x_1 − 1) + 1, rank_1(B, x_2)]
     return ORQ(B_0, [x_{1,0}, x_{2,0}], [y_1, y_2], [m, tmp]) + ORQ(B_1, [x_{1,1}, x_{2,1}], [y_1, y_2], [tmp, M])
end function
ORQ(B, x_1, x_2, y_1, y_2, m, M)
```

Théorème 2.6.1. *This algorithm is correct and has complexity in $\mathcal{O}(\log_2(n))$*

Démonstration. — The algorithm finds the maximum range of points that covers $[y_1, y_2]$. The answer is the points in it with first coordinate in $[x_1, x_2]$. From question 5., we are tracking accordingly those points as we go down the tree and thus the algorithm is correct.

— Everytime we go down in depth in the tree, we update the intervals to check in the children of the node considered. When arriving at a node sufficiently small, the answer is given, thus, this algorithm has complexity in $\mathcal{O}(\log_2(n))$

■