Longest Common Subsequence

Tarun Mohandas Daryanani

Pedro Miguel Lagüera Cabrera

Abián Torres Torres

Content

- **Introduction:**
    - Short Description
    - Properties
- **Implementation:**
    - Recursive
    - Top-Down
    - Bottom-Up
- **Analysis:**
    - Recursive
    - Bottom-Up
    - Conclusions

Description

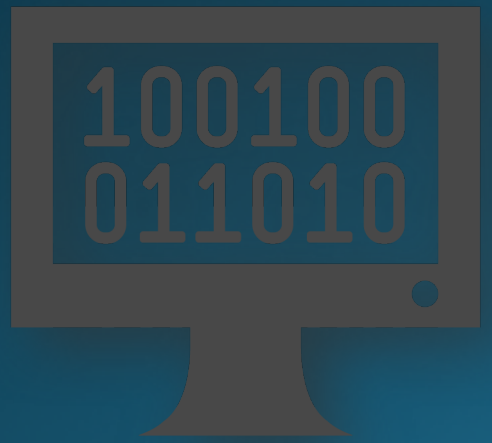
- LCS finds the longest subsequence common to all sequences in a set of sequences (often just two sequences).
- **Main applications**:
  - Bioinformatics
  - Revision Control Systems (Git)
- The LCS problem can be solved with **dynamic programmming**:
  - The problem can be broken down into smaller, simple "subproblems".
    - Which can be divided into yet simpler subproblems, until the solution becomes trivial.
  - The solution to high-level subproblems often reuse lower level subproblems.

Properties

- **First Property** – when both sequences end with the same element.
  1. To find their LCS, shorten each sequence by removing the last element.
  2. Find the LCS of the shortened sequences
  3. Append the removed element to the LCS.
- **Second Property** – both sequences end with different elements.
  - **First Case** – LCS ends like the first sequence but not like the second.
    - The last element of the second sequence can be removed.
    - **$LCS(X_n, Y_m) = LCS(X_n, Y_{m-1})$**
  - **Second Case** – LCS doesn't end like the first sequence.
    - The last element of the first sequence can be removed.
    - **$LCS(X_n, Y_m) = LCS(X_{n-1}, Y_m)$**

Implementations

- The LCS algorithm can be implemented through **three different approaches**.
  - A **Recursive** brute-force algorithm.
  - A recursive **top-down** algorithm.
  - An iterative **bottom-up** algorithm.



Implementation: Recursive

- It does not have any type of memory
- It usually solves subproblems already resolved
- Runs all possible combinations

```
private static int subproblem(int i, int j) {
        Recursive.setnOperations(Recursive.getnOperations()+1);
        if(i == -1 || j == -1)
                return 0;
        if((i >= 0) && (j >= 0) && (getA().charAt(i) == getB().charAt(j)))
                return subproblem(i - 1, j - 1) + 1;
        else
                return Math.max(subproblem(i - 1, j), subproblem(i, j - 1));
}
```

Implementation: Top Down

```
public int lcs() {
        setnOperations(0);
        for (int i = 0; i < table.getN(); i++)
            for (int j = 0; j < table.getM(); j++)
                table.set(i, j, -1);
        return subproblem(0, 0);
}
```

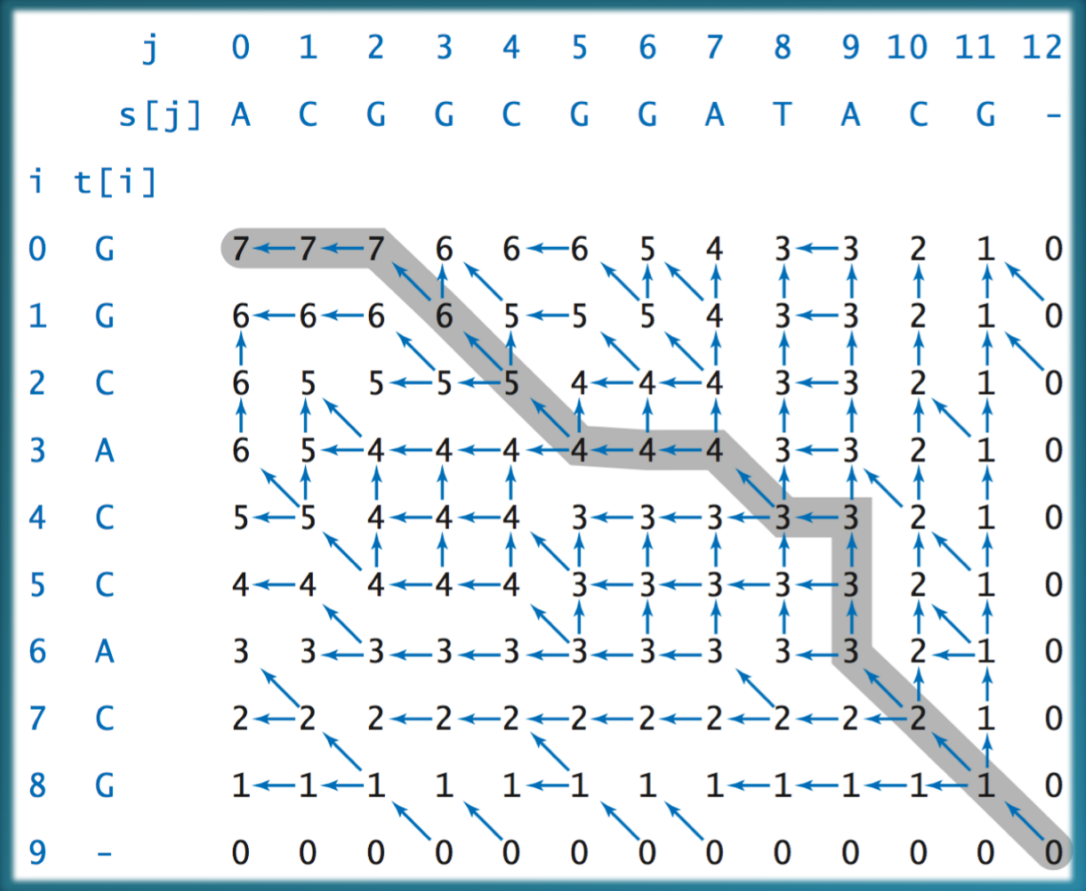
```
private static int subproblem(int i, int j) {
        setnOperations(getnOperations()+1);
        if(table.get(i, j) < 0){
                if(i == a.length() || j == b.length())
                        table.set(i, j, 0);
                else if(a.charAt(i ) == b.charAt(j))
                        table.set(i, j, subproblem(i + 1, j + 1) + 1);
                else
                        table.set(i, j, Math.max(subproblem(i + 1, j), subproblem(i, j + 1)));
        }
        return table.get(i, j);
}
```

- Similar to Recursive
- This time we start in i = 0 and j = 0.

- We add a memoization, so we do not solve a subproblem that was previously solved

Implementation: Top Down

- Backtracking:




```
String s = new String();
int i = 0;
int j = 0;
while (i < table.getN()-1 && j < table.getM()-1){
        if (getA().charAt(i)==getB().charAt(j)){
            s += getA().charAt(i) + "";
            i++; j++;
        }
        else if (table.get(i+1, j) > table.get(i, j+1)) {
                i++;
        }
        else {
                j++;
        }
}
    return s;
```

Implementation: Bottom Up

```
for (int i = 0; i < super.getTable().getN(); i++)
        for (int j = 0; j < super.getTable().getM(); j++){
                if(i == 0 || j == 0)
                        table.set(i, j, 0);
        }
```

- In this case we implement an iterative algorithm.
- We still use a memorization

```
for (int i = 1; i <= super.getA().length(); i++){
        for (int j = 1; j <= super.getB().length(); j++){
                if(super.getA().charAt(i-1) == super.getB().charAt(j-1))
                        table.set(i, j, table.get(i - 1, j - 1) + 1);
                else
                        table.set(i, j, Math.max(table.get(i - 1, j), table.get(i, j - 1)));
                setnOperations(getnOperations()+1);
        }
}
```

Implementation: Bottom Up

- Backtracking:

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	4
7	Z	0	1	2	2	3	3	3	4

```
StringBuffer sb = new StringBuffer();
for (int x = getA().length()-1, y = getB().length()-1;x != 0 && y != 0; ) {
        if (table.get(x, y) == table.get(x - 1, y))
                x--;
        else if (table.get(x, y) == table.get(x, y - 1))
                y--;                                    else {
                assert a.charAt(x-1) == b.charAt(y-1);
                sb.append(a.charAt(x-1));
                x--;
                y--;
        }
}
return new StringBuilder(sb.toString()).reverse().toString();
```

Analysis: Recursive

- Suppose we are in the worst case: Both strings are totally different.

A = “123456789” B = “abcdefghi”

- What action will the algorithm take on each recursive call, before de base cases?

```
int subproblem(int i, int j) {
  if(i == -1 || j == -1)
    return 0;
  if((i >= 0) && (j >= 0) && (A.charAt(i) == B.charAt(j)))
    return subproblem(i - 1, j - 1) + 1;
  else
    return Math.max(subproblem(i - 1, j), subproblem(i, j - 1));
}
```

Analysis: Recursive

- In each call, the algorithm will, in turn, make two recursive calls, until reaching the base cases.

```
return Math.max(subproblem(i - 1, j), subproblem(i, j - 1));
```

- In addition, if we assume that the lengths of both strings are equal, because in larger dimensions, the variation in size is insignificant.

- Therefore, we can deduce that the complexity, in upper bound will be:

$$O(2^n)$$

Analysis: Bottom-Up

- But, if we realize, we are recalculating subproblems, therefore, we will have to apply some algorithm of memoization. At this point, dynamic programming comes into play.

- Analyzing the bottom-up algorithm described previously, we see that, for any pair of strings, we will always have to traverse the matrix in its entirety. Consequently, we will have $(n + 1)(m + 1)$ operations:

$$O(nm)$$

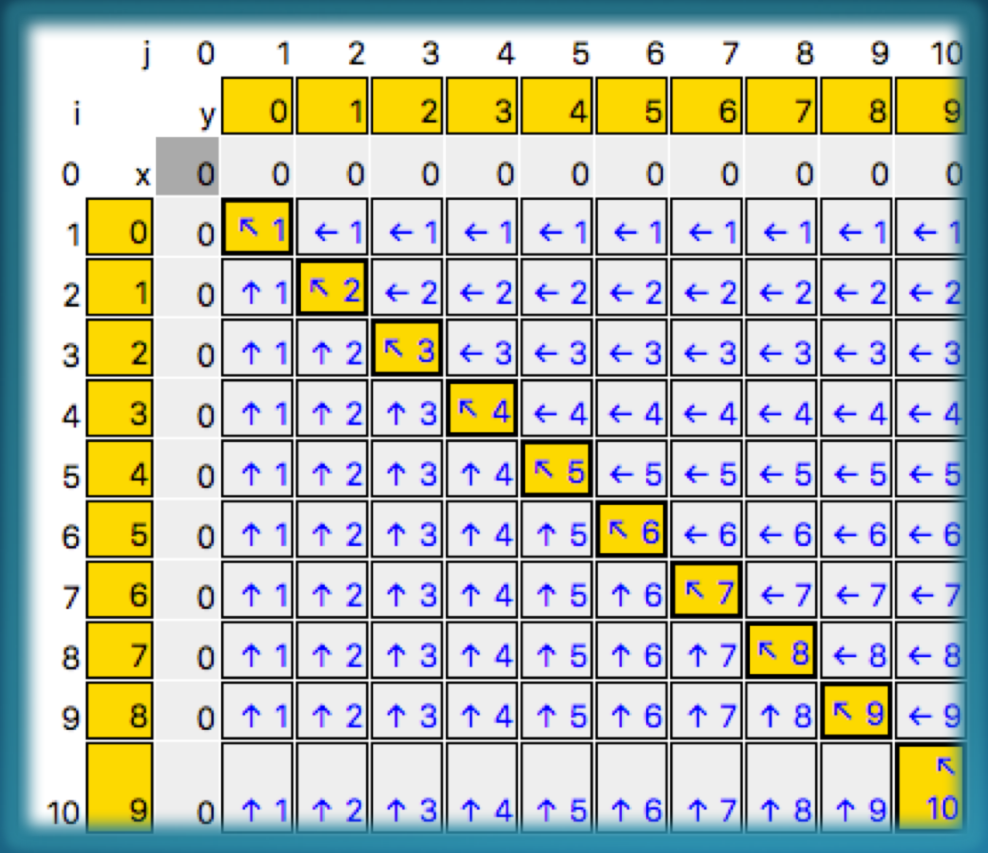
Analysis: Bottom-Up

- We still have to rebuild the solution. In the worst case, both strings are totally differents. Therefore, we will have to carry out $(n + m)$ operations. (We must remember that we can only make movements to the left and up)

	j	0	1	2	3	4	5	6	7	8	9	10
i		y	a	b	c	d	e	f	g	h	i	j
0	x	0	0	0	0	0	0	0	0	0	0	0
1	0	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
2	1	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
3	2	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
4	3	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
5	4	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
6	5	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
7	6	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
8	7	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
9	8	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
10	9	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0

Analysis: Bottom-Up

- And consequently, if both strings are equal, it will suffice to traverse the main diagonal of our matrix, what implies n operations.



	j	0	1	2	3	4	5	6	7	8	9	10
i		y	0	1	2	3	4	5	6	7	8	9
0	x	0	0	0	0	0	0	0	0	0	0	0
1	0	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2	1	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3	2	0	↑ 1	↑ 2	↖ 3	← 3	← 3	← 3	← 3	← 3	← 3	← 3
4	3	0	↑ 1	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4	← 4	← 4
5	4	0	↑ 1	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	← 5	← 5	← 5
6	5	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6	← 6	← 6
7	6	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↖ 7	← 7	← 7	← 7
8	7	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 7	↖ 8	← 8	← 8
9	8	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 7	↑ 8	↖ 9	← 9
10	9	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 7	↑ 8	↑ 9	↖ 10

Analysis: Conclusions

- We obtain that, applying dynamic programming, we reduce considerably the complexity described in the recursive case, then:
  - Bottom-Up:

$$\Omega((nm) + (n))$$

$$O((nm) + (n + m)))$$

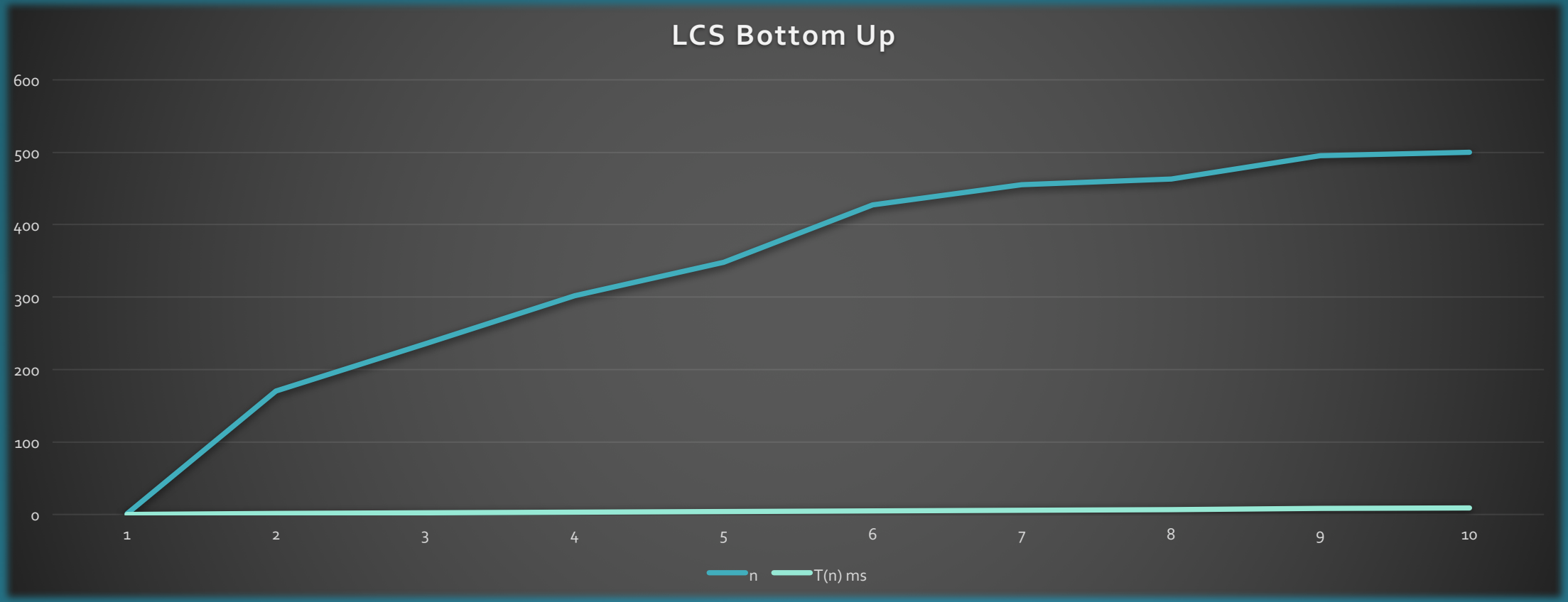
  - Recursive:

$$O(2^n)$$

Analysis: Conclusions

- And to verify what we have studied, let's prove it experimentally (Depends slightly on the computational resources used):

Analysis: Conclusions



Analysis: Conclusions

- But we can still further polish the efficiency of this algorithm:
  - Parallelism. Spread the work in multiple cores. So, if we have 4 cores, the temporal complexity would be (Top-Down):

$$O(\frac{nm}{4})$$

  - Increasing space efficiency (Hirschberg's algorithm): If we carefully analyze the algorithm, we will realize that we only need the previous and current row.

$$O(2\min(n,m))$$

Bibliography

https://en.wikipedia.org/wiki/Method_of_Four_Russians
https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
http://lcs-demo.sourceforge.net/
https://www.ics.uci.edu/~eppstein/161/960229.html
https://en.wikipedia.org/wiki/Hirschberg%27s_algorithm

Thanks for your attention