Gorilla or Sea Cucumber (?) implementation

Arleth, Johan
Bele, Claudiu
Jacobsen, Andreas
Ulrik, Kenneth Ry

October 3, 2016

1 Results

When running the program it will output the input strings, the output string and the cost. The results of running the program will look as in figure 1, where the output of Human vs Human-sickle is generated.

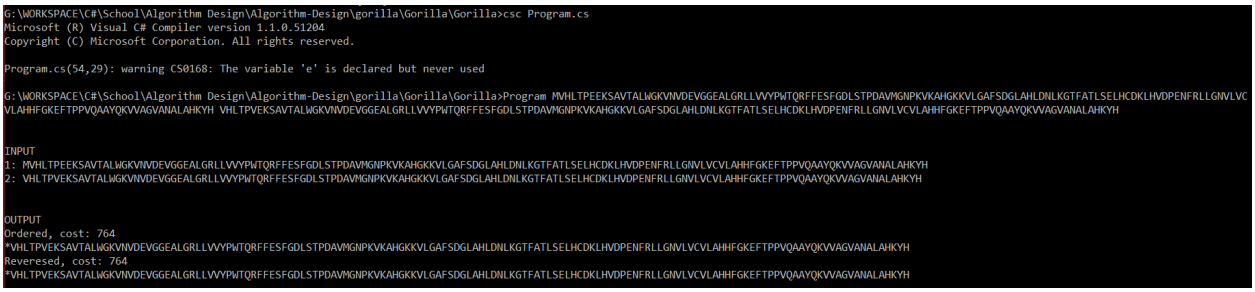

Figure 1: Command-line session of running the Human vs. Human-sickle comparison with our implementation.

Running this on obvious results as fx human and gorilla vs human and sea-cucumber, should hopefully produce a closer match between human and gorilla than between human and sea-cucumber. The result produced by human and gorilla is the cost of 777 (higher is better), and the combined string of:

MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMG-NPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLA-HHFGKEFTPPVQAAYQKVVAGVANALAHKYH

When running the human and sea-cucumber cost should be lower, and fortunately it is as it results in a cost of 80, and the combined string of:

****M*V**H**LTPEEKSAVTALWGK*V*NVDEVGGEALGRLL*VY*PWTQRFFESFGD-LST*DAVMGNPKVKAHGKKVLGAFSDGLAHLD*N*LKGTFATLSELH*D*LH*VDPENFRL-LGNVLVCVLAHHFGKEFTPPVQAAYQKVVA*VANA*LAHKYH

This tells us that we as humans are closer to gorillas than sea-cucumbers.

When we run the program on all the inputs we get the following table:

	Hu	HS	Go	SM	Ho	De	Pi	Co	Gu	Tr	Ro	La	Se
Hu	780	764	777	736	641	562	646	640	532	414	341	349	80
HS	764	774	761	733	638	555	643	633	529	419	350	120	75
Go	777	761	780	733	638	559	643	643	529	411	352	127	80
SM	736	733	733	770	655	572	644	647	537	410	347	121	73
Ho	641	638	638	655	768	540	634	615	535	395	361	101	83
De	562	555	559	572	540	756	565	608	464	360	328	111	59
Pi	646	643	643	644	634	565	761	623	534	375	378	126	85
Co	640	633	643	647	615	608	623	752	487	379	344	121	70
Gu	532	529	529	537	535	464	534	487	767	435	380	114	95
Tr	414	419	411	410	395	360	375	379	435	735	442	112	84
Ro	341	350	352	347	361	328	378	344	380	442	766	104	62
La	349	120	127	121	101	111	126	121	114	112	104	750	130
Se	80	75	80	73	83	59	85	70	95	84	62	130	785

Unfortunately the cost of Human-Lamprey is higher than expected. Besides that we get expected results on the rest of the string pairs.

2 Implementation details

2.1 Recursive Algorithm

The algorithm works by calculating all 3 possible costs between characters at specific indexes in the two strings we input. The three possible resolutions for characters at specific indexes in the two strings are:

- the cost of entering a white space instead of the first string's current character (also decreasing the first string's index by 1);
- the cost of exchanging the first string's character at its current index with the second string's character at its current index (also decreasing both current indexes by 1);
- the cost of entering a white space instead of the second string's current character (also decreasing the second string's index by 1)

The recursive calls will be done in the **DoWork** function, which takes two parameters indicating the current index of the two strings we need to calculate the cost for. The return type of

the method is a Tuple of int, String containing the **best** cost for the current String. We will be traversing the Strings from the end towards the beggining (i.e. passing index -1 being the base case, which returns a cost of "0" and an empty String).

After the 3 recursive calls are done, the best cost is calculated as the maximum cost between all 3 Tuple<int,string> returns. After having the best cost, we append a character to the string from the tuple that has the best result and return a tuple containing the best cost and the appended string. The character we appent can either be an empty character (star symbol in our case) or the first string's current character (from replacing the characters).

2.2 Caching

What we are caching is the best result for two specific strings at specific indexes. This is either done at the end of the **DoWork** function after calculating a new best cost for a particular substring, or at the beggining where we have a special case when both indexes are 0.

At the beggining of the function we use **DoWork**'s parameters in order to try get the cached result for those two indexes. If a result is found, we return the cached result instead of making any more recursive calls.

2.3 Setup

We initially expect the user to enter two different arguments when running the code. If the two values are not found in **main**'s parameters, we will make the calculation using the **gorilla** and **sea-cucumber** proteins. After receving user input, we prepend empty characters (star symbol) to one of the strings until their lengths match.