PROBLEM 1 – Match

Adrian Budău
University of Bucharest

Let's recall the traditional algorithm for checking whether a bracket string is a *valid bracket sequence*. The algorithm iterates over all characters sequentially and makes use of a stack. If the next character to process is an opening bracket, we push it to the top of the stack. If it is a closing bracket, we check the state of the stack. If it is empty, the sequence is not valid. Otherwise, we erase its top item. If at the end of the process the stack is non-empty, again, the sequence is not valid. Note that this also gives us a way to identify "matching" brackets (in the sense defined in the problem statement). Every opening bracket is matched with the closing bracket that "popped" it off the top of the stack.

Solution 1 – 11 points

It's reasonable to assume that this stack-based approach is valuable to our current problem. We act much in the same manner, processing the characters of string `S` from left to right: if the stack is empty, the current character is pushed to the top of the stack, acting as an opening bracket. The same happens if the stack is not empty but its topmost character differs from our current one. However, if the stack's topmost character is the same as our current one, then we can also opt to convert the current one into a closing bracket and "pop" the other one off the stack. It's not clear how a particular decision affects future options or the existence of a solution, so the safest way to take care of this issue is to backtrack through all possible sequences of decisions and retain the best valid bracket sequence, or detect that none exists. Because each character leaves us with at most 2 options, the time complexity of this algorithm is bounded by $O(2^N)$. It should score 11 points.

Solution 2 – 39 points

Going further, let's find a faster algorithm, but for a simpler problem. How fast can we decide if *any* matching valid bracket sequence exists for the given string? Intuitively, it seems that the only type of "bad" decision one can make is to open a bracket instead of closing an existing one. This is because the only way the algorithm can fail is by not having an empty stack at the end. It turns out that, indeed, closing brackets as soon as possible is a valid way of finding a solution if one exists.

To prove this, consider a scenario in which there exists a solution that was not found by this algorithm and show that it can be transformed into another solution that this algorithm will find. Therefore, we have a simple $O(N)$ algorithm that can check if a valid bracket sequence exists. Let's name this algorithm `DECIDE`. We can then apply a classical technique for obtaining lexicographically minimal solutions for a certain problem. For each decision we are confronted with, let's try to greedily convert the character into an opening bracket. To check if this might be a bad decision, run `DECIDE` on the current stack and the remaining string. If `DECIDE` tells us that it can no longer find any valid solution, we should instead choose to convert the character into a closing bracket. This gives us a simple $O(N^2)$ algorithm for our original problem. It should score 39 points.

Solution 3 – 100 points

Now let's look for a faster algorithm. Note that the first character will always be an open bracket. Let's assume we already *magically* know where its matching closing bracket will be in the optimal solution. Let's denote this value by `OptMatchBeginning`. Then, we can set the `0`-th character of the solution to be `'('`, the `OptMatchBeginning`-th character to be `')'` and then solve `S[1...OptMatchBeginning - 1]` and `S[OptMatchBeginning + 1...N - 1]` recursively.

This would be handy indeed, so let's see what `OptMatchBeginning` is and how we can find it quickly. For a certain position, `Pos`, to be a valid candidate, it should hold that `S[Pos] = S[0]` and `DECIDE(S[1..Pos - 1]) = True` (why?). Acting again on intuition, we conjecture that among these valid positions, the right-most one is the optimal one. To prove this, suppose `PosLeft` and `PosRight` are two candidates, with `PosLeft < PosRight`. We know the following:

```
DECIDE(S[1...PosLeft - 1]) = True        (1)
DECIDE(S[1...PosRight - 1]) = True       (2)
```

We also introduce

Lemma 1: If `DECIDE(S[x..z]) = True` and `DECIDE(S[x..y]) = True`, with `y < z`, then it holds that `DECIDE(S[y + 1..z]) = True`.

Proof hint: `DECIDE` behaves identically for `S[x..z]` and `S[x..y]` up to and including position `y`. Also, because `S[x..y]` is solvable, it means that immediately after position `y`, the stack is empty. It will also be empty after position `z`.

Applying *Lemma 1* on `(1)` and `(2)`, we deduce that `DECIDE(S[PosLeft..PosRight - 1]) = True`. We can infer that matching `S[0]` with `S[PosRight]` gives us a strictly better solution than matching it with `S[PosLeft]`. This is because the solution for `S[1..PosLeft - 1]` can remain exactly the same in both instances, but in the former case `S[PosLeft]` can become an opening bracket instead of a closing one.

We still have to make sure we can find `OptMatchBeginning` quickly in each recursive call. We'll pose the problem in such a way that it lends itself to an easy precomputation. We state the following:

Lemma 2: `OptMatchBeginning` is equal to the maximum position `X` for which `DECIDE(S[X + 1..N - 1]) = True` and `S[X] = S[0]`.

Proof hint: Using *Lemma 1* (we actually apply it on reversed strings, but it's easy to see everything stays the same) we deduce that `DECIDE(S[0..X]) = True`. The last thing we should prove is that `DECIDE(S[1..X - 1]) = True` (so `S[0]` can really be matched with `S[X]`). Imagine any solution for `DECIDE(S[0..X])`. If `S[0]` and `S[X]` are not matched with each other, but with, say, `S[Y]` and `S[Z]` respectively, we can safely match `S[Y]` and `S[Z]` together and then match `S[0]` with `S[X]` keeping the rest of the solution exactly the same.

Now that *Lemma 2* is proven true, we'll precompute the following table: `prevStart[i][c]` = the maximum position `j`, with `j ≤ i` such that `DECIDE(S[j + 1..i]) = True` and `S[j] = c`. This precomputation needs `O(N * SIGMA)` memory and `O(N * SIGMA)` time (its recurrence is left as an exercise). Then, `OptMatchBeginning` is simply equal to `prevStart[N - 1][S[0]]`. It may not be apparent that doing this precomputation for the original string is enough to correctly determine `OptMatchBeginning` values for *all* recursive calls. Specifically, say we're solving `S[left..right]` recursively; is it always true that `prevStart[right][S[left]] > left`? This is indeed the case, but if you're unconvinced, try proving it formally. The reasoning is similar to our previous proof outlines.

The described algorithm has `O(N * SIGMA)` time complexity and should score 100 points. Purely linear solutions also exist, we invite you to find them!