Special Binary String

Special binary strings are binary strings with the following two properties:

- The number of 0's is equal to the number of 1's.
- Every prefix of the binary string has at least as many 1's as 0's.

Given a special string `S`, a *move* consists of choosing two consecutive, non-empty, special substrings of `S`, and swapping them. *(Two strings are consecutive if the last character of the first string is exactly one index before the first character of the second string.)*

At the end of any number of moves, what is the lexicographically largest resulting string possible?

Example 1:

```
Input: S = "11011000"
Output: "11100100"
Explanation:
The strings "10" [occuring at S[1]] and "1100" [at S[3]] are swapped.
This is the lexicographically largest string possible after some number of swaps.
```

Note:

1. `S` has length at most `50`.
2. `S` is guaranteed to be a *special* binary string as defined above.

Solution 1

Just 4 steps:

1. Split S into several special strings (as many as possible).
2. Special string starts with 1 and ends with 0. Recursion on the middle part.
3. Sort all special strings in lexicographically largest order.
4. Join and output all strings.

Updated:

The middle part of a special string may not be another special string. But in my recursion it is.
For example, 1M0 is a splitted special string. M is its middle part and it must be another special string.

Because:

1. The number of 0's is equal to the number of 1's in `M`
2. If there is a prefix `P` of `M` which has one less 1's than 0's, `1P` will make up a special string. `1P` will be found as special string before `1M0` in my solution. It means that every prefix of `M` has at least as many 1's as 0's.

Based on 2 points above, M is a special string.

C++ version

```
string makeLargestSpecial(string S) {
    int count = 0, i = 0;
    vector<string> res;
    for (int j = 0; j < S.size(); ++j) {
      if (S[j] == '1') count++;
      else count--;
      if (count == 0) {
        res.push_back('1' + makeLargestSpecial(S.substr(i + 1, j - i - 1)) + '0');
        i = j + 1;
      }
    }
    sort(res.begin(), res.end(), greater<string>());
    string res2 = "";
    for (int i = 0; i < res.size(); ++i) res2 += res[i];
    return res2;
  }
```

Java version:

```java
public String makeLargestSpecial(String S) {
        int count = 0, i = 0;
        List<String> res = new ArrayList<String>();
        for (int j = 0; j < S.length(); ++j) {
          if (S.charAt(j) == '1') count++;
          else count--;
          if (count == 0) {
            res.add('1' + makeLargestSpecial(S.substring(i + 1, j)) + '0');
            i = j + 1;
          }
        }
        Collections.sort(res, Collections.reverseOrder());
        return String.join("", res);
    }
```

Python version:

```python
def makeLargestSpecial(self, S):
        count = i = 0
        res = []
        for j, v in enumerate(S):
            count = count + 1 if v=='1' else count - 1
            if count == 0:
                res.append('1' + self.makeLargestSpecial(S[i + 1:j]) + '0')
                i = j + 1
        return ''.join(sorted(res)[::-1])
```

Upvote and enjoy.

written by lee215 original link here

Solution 2

According to the description , there are 2 requirement for `Special-String`

1. The number of 0's is equal to the number of 1's.
2. Every prefix of the binary string has at least as many 1's as 0's.

The 2nd definition is essentially saying that at any point of the string, you cannot have more 0's than 1's.

If we map `'1'` to `'('`, `'0'`s to `')'`, a `Special-String` is essentially `Valid-Parentheses`, therefore share all the properties of a `Valid-Parentheses`
A `VP` (`Valid-Parentheses`) have 2 form:

1. single nested `VP` like `"(())"`, or `"1100"`;
2. a number of consecutive `sub-VP`s like `"()(())"`, or `"101100"`, which contains `"()" + "(())"` or `"10" + "1100"`

And this problem is essentially ask you to reorder the `sub-VP`s in a `VP` to make it bigger. If we look at this example : `"()(())"` or `"101100"`, how would you make it bigger?
Answer is, by moving the 2nd sub-string to the front. Because deeply nested VP contains more consecutive `'('`s or `'1'`s in the front. That will make reordered string bigger.

The above example is straitforward, and no recursion is needed. But, what if the groups of `sub-VP`s are not in the root level?
Like if we put another `"()(())"` inside `"()(())"`, like `"()(( (()(()) ))"`, in this case we will need to recursively reorder the children, make them `MVP` (`Max-Valid-Parentheses`), then reorder in root.

To summarize, we just need to reorder all groups of `VP`s or `SS`'s at each level to make them `MVP`, then reorder higher level `VP`s;

```
class Solution {
public:
    string makeLargestSpecial(string s) {
        int i = 0;
        return dfs(s, i);
    }

private:
    string dfs(string& s, int& i) {
        string res;
        vector<string> toks;
        while (i < s.size() && res.empty()) {
            if (s[i++] == '1') toks.push_back(dfs(s, i));
            else res += "1";
        }
        bool prefix = res.size();
        sort(toks.begin(), toks.end());
        for (auto it = toks.rbegin(); it != toks.rend(); it++) res += *it;
        if (prefix) res += '0';
        return res;
    }
};
```

written by alexander original link here

Solution 3

```
class Solution {
    public boolean isSpecial(String S){
        boolean res = true;
        int count = 0;
        for(int i = 0; i < S.length(); i++){
            if(S.charAt(i) == '1'){
                count++;
            } else {
                count--;
            }
            if(count < 0){
                return false;
            }
        }
        return count == 0;
    }

    public String swap(String s, int i, int j, int k, int l) {
        StringBuilder sb = new StringBuilder();
        for(int index = 0; index < i; index++) {
            sb.append(s.charAt(index));
        }
        for(int index = k; index <= l; index++) {
            sb.append(s.charAt(index));
        }
        for(int index = i; index <= j; index++) {
            sb.append(s.charAt(index));
        }
        for(int index = l+1; index < s.length(); index++) {
            sb.append(s.charAt(index));
        }
        return sb.toString();
    }

    public String makeLargestSpecial(String S) {
        if(S.length() == 0){
            return "";
        }

        String res = S;
        for(int i = 0; i < S.length(); i++){
            for(int j = i+1; j <= S.length(); j++){
                String curr = S.substring(i, j);
                if(isSpecial(curr)){
                    for(int k = j+1; k <= S.length(); k++){
                        String sec = S.substring(j, k);
                        if(isSpecial(sec)){
                            String str = swap(S, i, j-1, j, k-1);
                            if(str.compareTo(res) > 0){
                                res = str;
                            }
                        }
                    }
                }
            }
        }
```

```
        }

        if(res.equals(S)){
            return res;
        } else {
            return makeLargestSpecial(res);
        }
    }
}
```

written by simonzhu91 original link here

From Leetcoder.