```
        return 2 * res[0][0] + res[1][0];
}

public int[][] matrixPower(int[][] m, int p) {
        int[][] res = new int[m.length][m[0].length];
        for (int i = 0; i < res.length; i++) {
                res[i][i] = 1;
        }
        int[][] tmp = m;
        for (; p != 0; p >>= 1) {
                if ((p & 1) != 0) {
                        res = muliMatrix(res, tmp);
                }
                tmp = muliMatrix(tmp, tmp);
        }
        return res;
}

public int[][] muliMatrix(int[][] m1, int[][] m2) {
        int[][] res = new int[m1.length][m2[0].length];
        for (int i = 0; i < m2[0].length; i++) {
                for (int j = 0; j < m1.length; j++) {
                        for (int k = 0; k < m2.length; k++) {
                                res[i][j] += m1[i][k] * m2[k][j];
                        }
                }
        }
        return res;
}
```

拼接所有字符串产生字典顺序最小的大写字符串

【题目】

给定一个字符串类型的数组 strs，请找到一种拼接顺序，使得将所有的字符串拼接起来组成的大写字符串是所有可能性中字典顺序最小的，并返回这个大写字符串。

【举例】

strs=["abc"，"de"]，可以拼成"abcde"，也可以拼成"deabc"，但前者的字典顺序更小，所以返回"abcde"。

strs=["b"，"ba"]，可以拼成"bba"，也可以拼成"bab"，但后者的字典顺序更小，所以返回"bab"。

【难度】

校 ★★★☆

【解答】

有一种思路为：先把 strs 中的字符串按照字典顺序排序，然后将串起来的结果返回。这么做是错误的，比如题目中的例子 2，按照字典排序结果是 B、BA，串起来的大写字符串为"BBA"，但是字典顺序最小的大写字符串是"BAB"，所以按照单个字符串的字典顺序进行排序的想法是行不通的。如果要排序，应该按照下文描述的标准进行排序。

假设有两个字符串，分别记为 a 和 b，a 和 b 拼起来的字符串表示为 a.b。那么如果 a.b 的字典顺序小于 b.a，就把字符串 a 放在前面，否则把字符串 b 放在前面。每两个字符串之间都按照这个标准进行比较，以此标准排序后，再依次串起来的大写字符串就是结果。这样做为什么对呢？当然需要证明。

证明的关键步骤是证明这种比较方式具有传递性。

假设有 a、b、c 三个字符串，它们有如下关系：

a.b < b.a

b.c < c.b

如果能够根据上面两式证明出 a.c < c.a，说明这种比较方式具有传递性，证明过程如下：

字符串的本质是 *K* 进制数，比如，只由字符'a'~'z'组成的字符串其实可以看作 26 进制的数。那么字符串 a.b 这个数可以看作 a 这个数是它的高位，b 是低位，即 a.b=a*K 的 b 长度次方+b。举一个十进制数的例子，x=123，y=6789，x.y=x*10000+y=1230000+6789，其中，10000=10 的 4 次方，4 是 y 的长度。为了让证明过程便于阅读，我们把“K 的 b 长度次方”记为 k(b)。则原来的不等式可化简为：

a.b < b.a => a*k(b) + b < b*k(a) + a　　不等式 1

b.c < c.b => b*k(c) + c < c*k(b) + b　　不等式 2

现在要证明 a.c < c.a，即证明 a*k(c)+c < c*k(a)+a。

不等式 1 的左右两边同时减去 b，再乘以 c，变为 a*k(b)*c < b*k(a)*c+a*c-b*c。

不等式 2 的左右两边同时减去 b，再乘以 a，变为 b*k(c)*a + c*a - b*a < c*k(b)*a。

a，b，c 是 *K* 进制数，服从乘法交换律，有 a*k(b)*c == c*k(b)*a，所以有如下不等式：

b*k(c)*a + c*a-b*a < c*k(b)*a == a*k(b)*c < b*k(a)*c + a*c - b*c

=> b*k(c)*a + c*a - b*a < b*k(a)*c + a*c-b*c

=> b*k(c)*a - b*a < b*k(a)*c - b*c

=> a*k(c) - a < c*k(a) - c

=> a*k(c) + c < c*k(a) + a

即 a.c < c.a，传递性证明完毕。

证明传递性后，还需要证明通过这种比较方式排序后，如果交换任意两个字符串的位置所得到的总字符串，将拥有更大的字典顺序。

假设通过如上比较方式排序后，得到字符串的序列为：

...A.M1.M2...M(n-1).M(n).L...

该序列表示，代号为 A 的字符串之前与代号为 L 的字符串之后都有若干字符串用“…”表示，A 和 L 中间有若干字符串，用 M1..M(n)。现在交换 A 和 L 这两个字符串，交换之前和交换之后两个总字符串就分别为：

...A.M1.M2...M(n-1).M(n).L... 换之前

...L.M1.M2...M(n-1).M(n).A... 换之后

现在需要证明交换之后的总字符串字典顺序大于交换之前的，具体过程如下。

在排好序的序列中，M1 排在 L 的前面，所以有 M1.L < L.M1，进一步有：

...L.M1.M2...M(n-1).M(n).A... > ...M1.L.M2...M(n-1).M(n).A...

在排好序的序列中，M2 排在 L 的前面，所以有 M2.L < L.M2，进一步有：

...M1.L.M2...M(n-1).M(n).A... > ...M1.M2.L...M(n-1).M(n).A...

在排好序的序列中，M(i)排在 L 的前面，所以有 M(i).L < L.M(i)，进一步有：

...M1.M2...L.M(i)...M(n-1).M(n).A... > ...M1.M2...M(i).L...M(n-1).M(n).A...

最终，...M1.M2...M(n-1).M(n).L.A... > ...M1.M2...M(n-1).M(n).A.L...

在排好序的序列中，A 排在 M(N)的前面，所以有 A.M(n) < M(n).A，进一步有：

...M1.M2...M(n-1).M(n).A.L... > ...M1.M2...M(n-1).A.M(n).L...

在排好序的序列中，A 排在 M(n-1)的前面，所以有 A.M(n-1) < M(n-1).A，进一步有：

...M1.M2...M(n-1).A.M(n).L... > ...M1.M2...A.M(n-1).M(n).L...

最终，...M1.A.M2...M(n-1).M(n).L... > ...A.M1.M2...M(n-1).M(n).L...

所以，...A.M1.M2...M(n-1).M(n).L... < ... < ...L.M1.M2...M(n-1).M(n).A...

解法有效性证明完毕。

那么整个解法的时间复杂度就是排序本身的复杂度，即 $O(N\log N)$。具体请参看如下代码中的 lowestString 方法。

```
public class MyComparator implements Comparator<String> {
```

```
        @Override
        public int compare(String a, String b) {
                return (a + b).compareTo(b + a);
        }
}

public String lowestString(String[] strs) {
        if (strs == null || strs.length == 0) {
                return "";
        }
        // 根据新的比较方式排序
        Arrays.sort(strs, new MyComparator());
        String res = "";
        for (int i = 0; i < strs.length; i++) {
                res += strs[i];
        }
        return res;
}
```

本题的解法看似非常简单，但解法有效性的证明却比较复杂。在这里不得不提醒读者，这道题的解题方法可以划进贪心算法的范畴，这种有效的比较方式就是我们的贪心策略。

正如本题所展示的一样，贪心策略容易大胆假设，但策略有效性的证明可就不容易求证了。在面试中，如果哪一个题目决定用贪心方法求解，则必须用较大的篇幅去证明你提出的贪心策略是有效的。所以建议面试准备时间不充裕的读者不要轻易去啃有关贪心策略的题目，那将占用大量的时间和精力。

在面试中，实际上也较少出现需要用到贪心策略的题目，造成这个现象有两个很重要的原因，其一是考查贪心策略的面试题目，关键点在于数学上对策略的证明过程，偏离考查编程能力的面试初衷。其二是纯用贪心策略的面试题，解法的正确性完全在于贪心策略的成败，而缺少其他解法的多样性，这样就会使这一类面试题的区分度极差，所以往往不会成为大公司的面试题。贪心策略在算法上的地位当然重要，但对初期准备代码面试的读者来说，性价比不高。

找到字符串的最长无重复字符子串

【题目】

给定一个字符串 str，返回 str 的最长无重复字符子串的长度。