```
def longestPalindrmStr(str):
	strList = []
	def substrLister(str, lst): #think that this is redundant, because of many “”
	#results, but I may be wrong
	#I believe this works
		If str == “”:
			return strList
		return substrLister(str[1:], lst + str[0]), substrLister(str[1:], lst)
	return  substrLister(str[1:], lst)

	def isPalindrm(str):
	# easy way to just do str[-1] == str
	#str[::-1] reverses any iterable, such as this string (a palindrome fits this definition
	#perfectly -- a palindrome is a string that is the same read from the front as read from
	#the back (i.e. “eye”)
	#but I want to code recursively by hand
		lengthStr = len(str) - 2
		def helper(str, length):
			if str[0] == str[-1]:
				return helper(str[1:] + str[lengthStr], lengthStr - 1) # some counting
	issue, with some bugs, but I hope this is ok
			elif str == “”:
				return True
			else:
			#not equal and not empty, then automatically its not a palindrome, so its
			#false
				return False
		helper(str, lengthStr)

	plndrmStrList = []
	for indSubstr in range(len(strList)):
		tmp = strList[indSubstr])
		if isPalindrm(tmp):
			plndrmStrList.append(tmp):

	#map(len(), plndrmStrList)

	for subStr in plndrmStrList:
		subStr = len(subStr)

	Return max(plndrmStrList)
```

```
#I did terribly on this coding interview
#I completely failed in writing this code
#I only had an unoptimized, algorithmic approach
#hard part is listing all possible substrings from a string (in an array)
        def listAllSubstr(string):
                substrLst = []
                def helper(string, stringBuild):
                        if string == “”:
                                substrLst.append(stringBuild)
                                stringBuild = “”
                        elif string
                        else:
                return helper(string, “”)
```