به نام خدا

تحلیل پیچیدگی زمانی بخش محاسبات ماشین حساب / 993613001 / Ali Ebrahimi

1. تابع isValid(infix)

این تابع رشته infix را کاراکتر به کاراکتر پیمایش می کند. هر مرتبه یک یا چند شرط ساده را روی آن کاراکتر و گاها کاراکتر قبلی یا بعدی آن چک می کند و اگر هر یک از شرط ها برقرار شود اجرای تابع تمام می شود. بنابرین پیچیدگی زمانی این تابع O(n) است.

```
private fun isValid(infix: String): Boolean {
    if (infix.isEmpty()) return false

    val len = infix.length
    val f = infix.first()
    val l = infix.last()
    val stack = MyStack<Char>(len)
    var operand = StringBuilder()

    if (!isDigit(f) && !isAngle(f) && f != '(' && f != '+' && f != '-' || !isDigit(l) && l != ')') {
        return false
    }
    for ((i, c) in infix.withIndex()) {

        if (isDigit(c)) {
            if (c == '.' && (len == 1 || "." in operand)) return false
            operand.append(c)

        } else {

            if (operand.isNotEmpty()) {
                if (operand.toString() == ".") return false
                operand = StringBuilder()
            }
            when {
                isOperator(c) -> {
                    if (i > 0 && isOperator(infix[i - 1])) return false
                }
                c == '(' -> {
                    stack.push(c)
                    if (i < len - 1 && infix[i + 1] in ")*/^") return false
                }
                c == ')' -> {
                    if (stack.isEmpty() || stack.pop() != '(') return false
                    if (i > 0 && isOperator(infix[i - 1])) return false
                }
                !isAngle(c) -> {
                    return false
                }
            }
        }
    }
    return stack.isEmpty()
}
```

2. تابع toPostfix(infixExpression)

این تابع در قسمت 1 روی رشته ورودی تغییراتی به کمک regex اعمال می کند. پیچیدگی این بخش O(n) است. سپس در قسمت 2 تابع isValid(infix) که قبلا بررسی کردیم با پیچیدگی O(n) اجرا می شود. پس از آن رشته infix را کاراکتر به کاراکتر پیمایش می کنیم. حین پیمایش یک یا چند شرط ساده روی کاراکتر چک می شود. به جز قسمت های3و 4 که حلقه های while وجود دارد. اما تعداد تکرار این حلقه ها محدود است زیرا تعداد اپراتور ها محدود است(مگر اینکه تعداد زیادی پرانتز باز پشت سر هم یا توان های متوالی مثل 2^3^4^5... داشته باشیم). بنابراین در اندازه های بزرگ تعداد تکرار این حلقه ها نسبت به سایز infix بسیار کمتر است و میتوان آن را عدد ثابت درنظر گرفت. پس پیچیدگی این بخش(پیمایش) نیز O(n) می شود. در نهایت در قسمت 5 هرچه در استک مانده را از آن پاپ میکنیم که طبق چیزی که گفته شد پیچیدگی آن را عدد ثابت در نظر میگیریم. بنابرین پیچیدگی کل تابع O(n) می شود.

```
private fun toPostfix(infixExpression: String): ArrayList<String> {
    var infix = infixExpression

    infix = infix
        .replace("\\s".toRegex(), replacement: "")
        .replace( oldValue: "π", newValue: "($PI)")
        .replace("sin".toRegex(), replacement: "s")
        .replace("cos".toRegex(), replacement: "c")
        .replace("tan".toRegex(), replacement: "t")
        .replace("([)])([\\d(.stc])".toRegex(), replacement: "$1*$2")
        .replace("([\\d.])([(sct])".toRegex(), replacement: "$1*$2")

    if (!isValid(infix)) throw RuntimeException("Error: wrong format")

    val len = infix.length
    val result = ArrayList<String>()
    var operand = StringBuilder()
    val operators = MyStack<Char>(len)

    for ((i, c) in infix.withIndex()) {

        // form operand by putting digits together
        if (isDigit(c)) {
            operand.append(c)

        } else {

            // add operand to result
            if (operand.isNotEmpty()) {
                result.add(operand.toString())
                operand = StringBuilder()
            }

            // arrange operators by priority
            when {
                isOperator(c) -> {
                    while (!operators.isEmpty() && !precedence(c, operators.peek())) {
                        result.add(operators.pop().toString())
                    }

                    // check - is neg not sub  or  + is not sum
                    if ((c == '-' || c == '+') && (i == 0 || infix[i - 1] == '(')) {
                        operators.push(if (c == '+') 'p' else 'n')
                    } else {
                        operators.push(c)
                    }
                }
                isAngle(c) -> {
                    operators.push(c)
                }
                c == '(' -> {
                    operators.push(c)
                }
                c == ')' -> {
                    while (!operators.isEmpty() && operators.peek() != '(') {
                        result.add(operators.pop().toString())
                    }
                    operators.pop()
                }
            }
        }
    }
    if (operand.isNotEmpty()) {
        result.add(operand.toString())
    }
    while (!operators.isEmpty()) {
        result.add(operators.pop().toString())
    }

    return result
}
```

1

2

3

4

5

3. تابع calculate(infix: String, steps: Boolean)

این تابع یک بولین در ورودی می گیرد و اگر true باشد کد های مربوط به قسمت تاریخچه را اجرا می کند. در تحلیل بخش محاسبات کاری با این قسمت ها نداریم چون بولین ورودی false است. این تابع در قسمت 1 تابع toPostfix(infixExpression) را که قبلا بررسی شد صدا می زند. پس پیچیدگی این بخش O(n) است. پس از آن آرایه postfix را پیمایش می کنیم. حین پیمایش بررسی میکنیم آیا رشته عدد هست یا خیر. تابع isNumber(string) این بررسی را به کمک ریجکس انجام میدهد که میتوان پیچیدگی آن را عدد ثابت فرض کرد (فرض میکنیم طول رشته مورد بررسی در برابر طول postfix ناچیز است) . اگر رشته مورد نظر عدد نبود قسمت elseاجرا می شود. در این قسمت نیز یک عملیات محاسباتی با پیچیدگی عدد ثابت انجام می شود. بنابرین این پیمایش O(n) هزینه دارد و پیچیدگی تابع O(n)می شود.

```
fun calculate(infix: String, steps: Boolean): Any {
    val operands = MyStack<String>(infix.length)

    if (steps) {
        history = ArrayList()
        history.add(infix)
        history.add(toInfix(toPostfix(infix)))
    }

    val postfix = toPostfix(infix)
    val newPostfix = ArrayList(postfix)
    var cur = 0

    for (c in postfix) {
        when {
            isNumber(c) -> {
                operands.push(c)
            }
            else -> {
                val o1 = operands.pop()

                val o2 = if (isOperator(c[0]))
                    operands.pop()
                else "0"

                val res = calculate(c[0], o1, o2)
                operands.push(
                    res.toString()
                )

                if (steps) {
                    newPostfix[cur] = format(res)

                    newPostfix.removeAt( index: cur-- - 1)

                    if (isOperator(c[0])) {
                        newPostfix.removeAt( index: cur-- - 1)
                    }

                    val step = toInfix(newPostfix)
                    if (history.last() != step)
                        history.add(step)
                }
            }
        }
        cur++
    }

    return if (steps)
        history
    else
        operands.pop().toDouble()
}
```

1