import kotlin.math.min

/\* Computes n choose k several ways & compares their running times. \*/

fun main(args: Array<String>) {

val n = 1300

val k = 700

testRun("dpChooseSpace", n, k, ::dpChooseSpace)

testRun("dpChooseRow", n, k, ::dpChooseRow)

testRun("dpChooseCol", n, k, ::dpChooseCol)

testRun("memChoose", n, k, ::memChoose)

testRun("naiveChoose", n, k, ::naiveChoose)

}

/\* Test harness. Calls f(n, k), then prints the result and running time. \*/

fun testRun(fname: String, n: Int, k: Int, f: (Int, Int) -> Int) {

var result = 0

val exeTime = kotlin.system.measureNanoTime { result = f(n, k) }

println("Running $fname($n, $k) gives $result and it takes $exeTime ns to execute.")

}

/\* Computes n choose k using the recurrence naively. \*/

fun naiveChoose(n: Int, k: Int): Int =

if (k == 0 || k == n) 1

else naiveChoose(n-1, k) + naiveChoose(n-1, k-1)

/\* Computes n choose k using bottom-up dynamic programming, filling

the table in row-major order. \*/

fun dpChooseRow(n: Int, k: Int): Int {

val table = Array<IntArray>(n+1) {

row -> IntArray(1 + min(k, row))

}

for (row in 0 until table.size)

for (col in 0 until table[row].size)

table[row][col] = if (col == 0 || col == row) 1

else table[row-1][col] + table[row-1][col-1]

return table[n][k]

}

/\* Computes n choose k using bottom-up dynamic programming, filling

the table in column-major order. \*/

fun dpChooseCol(n: Int, k: Int): Int {

val table = Array<IntArray>(n+1) {

row -> IntArray(1 + min(k, row))

}

for (col in 0..k)

for (row in col..n)

table[row][col] = if (col == 0 || col == row) 1

else table[row-1][col] + table[row-1][col-1]

return table[n][k]

}

/\* Computes n choose k using bottom-up dynamic programming,

space-saving version \*/

fun dpChooseSpace(n: Int, k: Int): Int {

val table = IntArray(1 + k)

for (row in 0..n)

for (col in min(row, k) downTo 0)

table[col] = if (col == 0 || col == row) 1

else table[col] + table[col-1]

return table[k]

}

/\* Computes n choose k using memoization. \*/

fun memChoose(n: Int, k: Int): Int {

val UNKNOWN = -1

val table = Array<IntArray>(n+1) {

row -> IntArray(1 + min(k, row)) {

col -> when (col) {

0, row -> 1

else -> UNKNOWN

}

}

}

fun accessTable(i: Int, j: Int): Int {

if (table[i][j] == UNKNOWN)

table[i][j] = accessTable(i-1, j) + accessTable(i-1, j-1)

return table[i][j]

}

return accessTable(n, k)

}