An Experiment on the Cognitive Complexity of Code

Michael Hansen (mihansen@indiana.edu)
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

Robert L. Goldstone (rgoldsto@indiana.edu)
Dept. of Psychological and Brain Sciences, 1101 E. 10th Street
Bloomington, IN 47405 USA

Andrew Lumsdaine (lums@indiana.edu)
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

Abstract

What simple factors impact the cognitive complexity of code? We present an experiment in which participants predict the output of ten small Python programs. Even with such simple programs, we find a complex relationship between code, expertise, and correctness. We use subtle differences between program versions to demonstrate that small notational changes can have profound effects on comprehension. We catalog common errors for each program, and perform an in-depth data analysis to uncover effects on response correctness and speed.

Keywords: program comprehension; psychology of programming; code complexity.

Introduction

What simple factors impact the cognitive complexity of code? A program is an abstract specification for a set of operations, but its code exists in a concrete notational system made to be read and written by humans. We expect programs with many complex operations to be hard to understand, but even operationally simple programs may be difficult when misleading variable names are used or implicit expectations are violated (Soloway & Ehrlich, 1984). It is common practice in some organizations to use "code complexity" metrics, such as number of lines and control flow statements, to identify potentially error-prone code that is hard to comprehend (El-Eman, 2001). Well-defined metrics, including lines of code (LOC) and cyclomatic complexity (CC) (McCabe, 1976), have recommended limits for large software projects (NDepend, 2013). One failing of these metrics is that they can not take a programmer's experience into account. While experience does not necessarily equal expertise, a veteran programmer and a novice are likely to face different challenges when comprehending the same code.

We present a web-based experiment in which participants with a wide variety of Python and overall programming experience predict the output of ten small Python programs. Most of the program texts are well within recommended LOC and CC limits (< 20 LOC, < 15 CC). Each program *type* has two or three *versions* with subtle differences that do not significantly disturb these complexity metrics. For each participant and program, we grade their text response on a 10 point scale, and record the amount of time taken. Using R, we perform an in-depth data analysis to answer the following research questions: (1) what common errors do participants make for different program types and versions?, (2) are some program versions more difficult than others?, and (3) what role does experience play in correctness and speed of response? Our results (summarized in Table 1) indicate that the relationship between program text, experience, and response is quite complex, even for these simple programs.

Related Work

Psychologists have been studying programmers for at least forty years. Early research focused on correlations between task performance and human/language factors, such as how the presence code comments impacts scores on a program comprehension questionnaire. More recent research has revolved around the cognitive processes underlying program comprehension. Effects of expertise, task, and available tools on program understanding have been found (Détienne & Bott, 2002). Studies with experienced programmers have revealed conventions, or "rules of discourse," that can have a profound impact (sometimes negative) on expert program comprehension (Soloway & Ehrlich, 1984).

Our present research focuses on programs much less complicated than the average professional programmer likely comprehends on a daily basis. The demands of our task are still high, however, since participants must predict precise program output. In this way, it is similar to debugging a short snippet of a larger program. Code studies often take the form of a code review, where programmers must locate errors or answer comprehension questions after the fact (e.g., does the program define a Professor class? (Burkhardt, Détienne, & Wiedenbeck, 2002)). Our task differs by asking programmers to mentally simulate code without necessarily understanding its purpose. We intentionally use meaningless identifier names where appropriate (`a`, `b`, etc.) to avoid influencing the programmer's mental model[1].

Similar research has asked beginning (CS1) programming students to read and write code with simple goals, such as the Rainfall Problem (Guzdial, 2011). To solve it, students

[1]For longer programs, such as `rectangle` and `between`, we use meaningful identifier/class names in order to focus attention on the differences between versions.

must write a program that averages a list of numbers (rainfall amounts), where the list is terminated with a specific value – e.g., a negative number or 999999. CS1 students perform poorly on the Rainfall Problem across institutions around the world, inspiring researchers to seek better teaching methods. Our work includes many Python novices with a year or less of experience (94 out of 162), so our results may contribute to the ongoing research in early programming education. We find that viewing vertical whitespace as irrelevant (`counting` program) and understanding pass-by-value semantics (`scope` program) are particularly challenging.

Methods

One hundred and sixty-two participants (129 males, 30 females, 3 unreported) were recruited from the Bloomington, IN area (29), on Amazon's Mechanical Turk (130), and via e-mail (3). All participants were required to have some experience with Python, though we welcomed beginners. The mean age was 28.4 years, with an average of 2.0 years of self-reported Python experience and 6.9 years of programming experience overall. Most of the participants had a college degree (69.8%), and were current or former Computer Science majors (52.5%). Participants from Bloomington were paid $10, and performed the experiment in front of an eye-tracker (see Future Work). Mechanical Turk participants were paid $0.75.

The experiment consisted of a pre-test survey, ten trials (one program each), and a post-test survey. Before the experiment began, participants were given access to a small Python "refresher," which listed the code and output of several small programs. The pre-test survey gathered information about the participant's age, gender, education, and experience. Participants were then asked to predict the printed output of ten short Python programs, one version randomly chosen from each of ten program types (Figure 1). The presentation order and names of the programs were randomized, and all answers were final. Although every program produced error-free output, participants were not informed of this fact beforehand. The post-test survey gauged a participant's confidence in their answers and the perceived difficulty of the task.

We collected a total of 1,602 trials from 162 experiments starting November 20, 2012 and ending January 19, 2013. Trials were graded semi-automatically using a custom grading program. A grade of 10 points was assigned for responses that exactly matched the program's output (1,007 out of 1,602 trials). A *correct* grade of 7-9 points was given when responses had the right numbers or letters, but incorrect formatting – e.g., wrong whitespace, commas, brackets. Common errors were given partial credit from 2 to 4 points, depending on correct formatting[2]. All other responses were manually graded by two graders whose instructions were to give fewer than 5 points for incorrect responses, and to take off points for incorrect formatting (clear intermediary calculations or comments were ignored). Graders' responses were strongly correlated ($r(598) = 0.90$), so individual trial grades were averaged. Trial reaction times ranged from 14 to 256 seconds. Outliers beyond two standard deviations of the mean (in log space) were discarded (60 of 1,602 trials). Participants had a total of 45 minutes to complete the entire experiment (10 trials + surveys), and were required to give an answer to each question.

[2]We considered an error common if three or more participants gave it as a response.

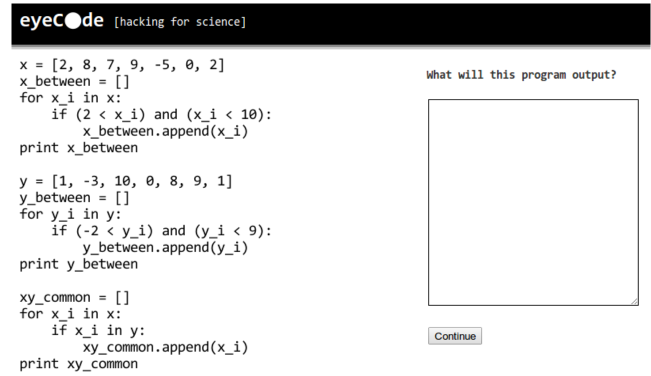

Figure 1: Sample trial from the experiment. Participants were asked to predict the output of ten Python programs.

We had a total of twenty-five Python programs in ten different categories. These programs were designed to be understandable to a wide audience, and therefore did not touch on Python features outside of a first or second introductory programming course. The programs ranged in size from 3 to 24 lines of code (LOC). Their cyclomatic complexities (CC) ranged from 1 to 7, and were medium-ly correlated with LOC ($r(25) = 0.46$, $p < 0.05$). CC was computed using the open source package PyMetrics.

Mechanical Turk One hundred and thirty participants were recruited from Mechanical Turk. Workers were required to have some Python experience, and could only participated once. All code was displayed as an image, making it difficult to copy/paste the code into a Python interpreter for quick answers. All responses were manually screened, and restarted trials or unfinished experiments were discarded.

Results

Data analysis was performed in R, and all regressions were done with R's built-in `lm` and `glm` functions. For linear regressions (grade and RT), we report intercepts and coefficients (β). For logistic regressions (probability of a correct answer or a common error), we report base predictor levels and the odds ratios (OR). Odds ratios can be difficult to interpret, and are often confused with relative risk (Davies, Crombie, & Tavakoli, 1998). While the direction of an effect is usually apparent, we caution their interpretation as effect sizes (especially when OR < 1).

Table 1 summarizes the results in terms of average grades, median reaction times (RT), and main effects (discussed in

detail below). Participants did well overall, though they had trouble with a few programs (e.g., `between`, `scope`, and `counting`). Programming experience did not significantly predict a participant's total grade in the experiment. This wasn't terribly surprising, since the programs were very simple, and a large number of trials resulted in perfect responses (1,007 out of 1,602). For individual trials, however, Python experience was a significant predictor of correctness probability – i.e., the probability of getting a grade of 7 or higher (base = 2.49, OR = 1.08, $p < 0.05$). We discuss grade and RT differences between program versions below.

between (2 versions) This program filters two lists of integers (`x` and `y`), prints the results, and then prints the numbers that `x` and `y` have in common. The `functions` version abstracts the `between` and `common` operations into reusable functions, while the `inline` version inlines the code, duplicating as necessary.

Since this is the longest and most complex program (in terms of CC), we expected more experienced programmers to be faster and to make fewer errors. We were surprised to find a significant effect of Python experience on the probability of making a very specific error (base = 0.13, OR = 1.44, $p < 0.5$). Instead of `[8, 9, 0]` for their last line of output, 22% of participants wrote `[8]` (for both versions). After the experiment, one participant said they mistakenly performed the "common" operation on lists `x_btwn` and `y_btwn` instead of `x` and `y` because it seemed like the next logical step. If others made the same mistake, this may suggest an addition to Soloway's Rules of Discourse (Soloway & Ehrlich, 1984): later computations should be the follow from earlier ones. We hypothesize that moving the `common` operation code before the two instances of `between` would eliminate this error.

counting (2 versions) This simple program loops through the range `[1, 2, 3, 4]`, printing "The count is `i`" and then "Done counting" for each number *i*. The **nospace** version has the "Done counting" `print` statement immediately following "The count is `i`," whereas the **twospaces** version has two blank lines in between. Python is sensitive to horizontal whitespace, but not vertical, so the extra lines do not change the meaning of the program.

We expected more participants to mistakenly assume that the "Done counting" `print` statement was not part of the loop body in the `twospaces` version. This was the case: 59% of responses in the `twospaces` version contained this error as opposed to only 15% in the `onespace` version (ref = `onespace`, OR = 4.0, $p < 0.0001$). Blank lines, while not usually syntactically relevant, have are positively correlated with code readability (Buse & Weimer, 2010). We did not find a significant effect of experience on the likelihood of making this mistake, suggesting that experts and novices alike may benefit from an ending delimiter (e.g., an `end` keyword or brackets).

funcall (3 versions) This program prints the product $f(1) * f(0) * f(-1)$ where $f(x) = x + 5$. The **nospace** version has no spaces between the calls to `f`, while the **space** version has a space before and after each `*`. The **vars** version saves the result of each call to `f` in a separate variable, and then prints the product of these variables.

Most people were able to get the correct answer (60) in 30 seconds or less. The most common errors (only 7% of responses) were 0, -60, and -80. We hypothesize that these correspond to the following calculation errors: assuming $f(0) = 0$, $f(-1) = -3$, and $f(-1) = -4$. There were no significant effects for grade or RT, so future versions of this experiment will likely exclude this program.

initvar (3 versions) The `initvar` program computes the product and sum of variables `a` and `b`. In the `good` version, `a` is initialized to 1, so the product computes $4! = 24$, and `b` is initialized to 0, making the summation 10. In the `onebad` version, $b = 1$, offsetting the summation by 1. In the `bothbad` version, $b = 1$ and $a = 0$, which makes the product 0.

We expected experienced programmers to make more errors due to the close resemblance of code in the `*bad` versions to common operations performed in the `good` version (factorial and summation). Instead, we found a significant negative effect of the `good` version on grade (intercept = 8.67, $\beta = -1.52$, $p < 0.05$), which is likely due to the difficulty of mentally performing 4 factorial. In the `bothbad` version, $a = 0$, allowing participants to short-circuit the multiplication (since a times anything is still zero). The `onebad` version, which also required performing the factorial, had a negative but non-significant effect on grade ($\beta = -0.97$).

order (2 versions) The `order` program prints the values of three functions, $f(x)$, $g(x)$, and $h(x)$. In the `inorder` version, f, g, and h are defined and called in the same order. The `shuffled` version defines them out of order (h, f, g).

We expected programmers to be slower when solving the `shuffled` version, due to an implicit expectation that function definitions and use would follow the same order. When including years of Python experience, we found a significant main effect on RT of the `shuffled` version (intercept = 54.3, $\beta = 21.0$, $p < 0.05$) as well as an interaction between experience and `shuffled` ($\beta = -7.1$, $p < 0.05$). These results indicate that having the functions defined out of order has a significant impact on reaction time, but that experience helps counter-act the effect to some degree.

overload (3 versions) This program uses the overloaded + operator, which serves as addition for integers and concatenation for strings. The `plusmixed` version uses both overloads of the operator, while the `multmixed` and `strings` versions only use + for string concatenation.

We expected programmers in the `plusmixed` version to make the mistake of interpreting `"5" + "3"` as 8 instead of `"53"` more often due to the priming of + as addition instead of concatenation. While this error occurred in about 11% of responses across all versions, we did not see a significant grade difference between versions. For reaction time, a significant interaction between overall programming experience and the `plusmixed` version was found (intercept = 42.5, $\beta = 3.34$, $p < 0.01$). This means that more experienced programmers took slightly longer on this version, perhaps due to the need

to recall the meaning of + for strings in Python.

partition (3 versions) The `partition` program iterates through the ranges $[1,4]$ (unbalanced) or $[1,5]$ (balanced), printing out `i low` for $i < 3$ and `i high` for $i > 3$. The `balanced` version outputs two `low` and two `high` lines, whereas the `unbalanced` versions produce two `low` lines and only one `high` line. The `unbalanced_pivot` version calls attention to 3 by assigning it to a variable named `pivot`.

We expected participants in the `unbalanced*` versions to add an additional `high` line because there were four numbers in the list (making it desirable for there to be four lines in the output). While there were a handful of responses like this, the most common error was simply leaving off the numbers on each line (e.g., `low` instead of `1 low`). Programmers seeing the `unbalanced` version were less susceptible to this error (ref = `balanced`, OR = 0.05, $p < 0.05$), though we saw no effect for the `unbalanced_pivot` version. More programming experience also helped participants avoid this kind of mistake across versions (base = 1.66, OR = 0.67, $p < 0.05$). We hypothesize that the `balanced` and `unbalanced_pivot` versions matched a "partition" schema for programmers, making them less likely to pay close attention to the loop body.

rectangle (3 versions) This program computes the areas of two rectangles using an `area` function with x and y scalar variables (`basic` version), (x,y) coordinate pairs (`tuples` version), or a `Rectangle` class (`class` version).

We expected participants seeing the `tuples` and `class` versions to take longer, since these versions contain more complicated structures. Almost everyone gave the correct answer, so there were no significant grade differences between versions. We found a significant RT main effect for the `tuples` version (intercept = 53.5, $\beta = 60.4$, $p < 0.01$), and an interaction between this version and Python experience ($\beta = -34.1$). This means that programmers in the `tuples` version took longer, but that additional Python experience helped reverse the effect. Surprisingly, we did not observe even a marginally significant RT effect for the `classes` version, despite it being the longest program of the three (21 lines vs. 14 and 18).

scope (2 versions) This program applies four functions to a variable named `added`: two `add_1` functions, and two `twice` functions. The `samename` version reused the name `added` for function parameters, while the `diffname` version used `num`. Because Python uses "pass by value" semantics with integers, and because neither of the functions return a value, `added` retains its initial value of 4 throughout the program (instead of being 22). This directly violates one of Soloway's Rules of Discourse (Soloway & Ehrlich, 1984): don't include code that won't be used.

We expected participants to mistakenly assume that the value of `added` was changed more often when the parameter names of `add_1` and `twice` were both also named `added` (`samename` version). There was **marginally** significant evidence for this ($p = 0.09$), but it was not conclusive. Additional Python experience helped reduce the likelihood of answering 22 (base = 1.28, OR = 0.71, $p < 0.05$), but around half of the participants still answered incorrectly.

whitespace (2 versions) The `whitespace` program prints the result of three simple linear calculations. In the `zigzag` version, the code is laid out as it flows, so that the line endings have a zig-zag appearance. The `linedup` version aligns each block of code by its operators.

We expected there to be a speed difference between the two versions in favor of `linedup`. When designing the experiment, most of our pilot participants agreed that this version was easier to read. The data did not support this claim, but there was a significant effect on the probability of not respecting order of operations. For the `zigzag` version, participants were significantly more likely to incorrectly answer 5, 10, and 15 for the y column (ref = `linedup`, OR = 0.04, $p < 0.05$). This suggests that when computing the y values, participants in the `zigzag` version did addition before multiplication more often. Effects of spacing on the perceived order of arithmetic operations has been studied before (Landy & Goldstone, 2010), and ours results indicate that code layout also has an impact.

Discussion

With the results of our experiment, we can begin to answer the research questions posed in the Introduction. First, what common errors do participants make for different program types and versions? Some errors were detail-oriented, such as leaving out the numbers on each line of output for the `partition` programs. Others, like the `[8]` error for `between` or the "Done counting" mistake on the `twospaces` version of `counting`, demonstrate an oversight due to strong expectations. In the `zigzag` version of the `whitespace` program, we also saw that participants were more likely to make calculation errors involving order of operations.

Second, are some program versions more difficult than others? In terms of average grade, the `between`, `counting`, and `scope` program types were the most difficult. Given LOC and CC metrics, this was expected for `between`, but not the others. These "difficult" programs simply tested fundamental concepts of the Python language, namely the relevance of horizontal whitespace (`counting`) and pass-by-value semantics for primitive types (`scope`). Technical arguments can be made for these concepts, but our results beg a deeper question: to what extent are humans considered in the design of programming languages (Hanenberg, 2010)?

Lastly, what role does experience play in correctness and speed of response? In several cases, both Python and overall programming experience aided in correctness and speed. Additional Python experience helped programmers avoid errors in the `scope` programs, but hurt for both versions of `between` (experienced programmers were more likely to assume later computations should include previous results). Less experienced programmers were slowed down in the `shuffled` version of `order`, but a got slight speed boost in the `plusmixed` version of `overload` (likely due to not having experience

with overloaded operators across many languages). In general, the relationship between experience, correctness, and speed turned out to be fairly complex.

Future Work During the course of the experiment, Bloomington participants were seated in front of a Tobii X300 eye-tracker. We plan to analyze this eye-tracking data, and correlate it with our findings here. Specifically, we hope to see how code features and experience effect the visual search process and, by proxy, program comprehension.

For future experiments, we would like to include other languages and more realistic programs (e.g., multiple files and modules). Our work to date has also focused exclusively on reading programs, so a similar experiment to this one where participants *write* short programs may prove insightful.

Acknowledgments

I would like to thank both the Percepts and Concepts Lab and the Center for Research in Extreme Scale Technologies (CREST) Lab for funding and use of equipment/facilities.

References

Burkhardt, J., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, *7*(2), 115–156.

Buse, R., & Weimer, W. (2010). Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, *36*(4), 546–558.

Davies, H. T. O., Crombie, I. K., & Tavakoli, M. (1998). When can odds ratios mislead? *Bmj*, *316*(7136), 989–991.

Détienne, F., & Bott, F. (2002). *Software design–cognitive aspects*. Springer Verlag.

El-Eman, K. (2001). Object-oriented metrics: A review of theory and practice. national research council canada. *Institute for Information Technology*.

Guzdial, M. (2011, February). From science to engineering. *Commun. ACM*, *54*(2), 37–39. Available from http://doi.acm.org/10.1145/1897816.1897831

Hanenberg, S. (2010). Faith, hope, and love: an essay on software science's neglect of human factors. In *Acm sigplan notices* (Vol. 45, pp. 933–946).

Landy, D., & Goldstone, R. L. (2010). Proximity and precedence in arithmetic. *The Quarterly Journal of Experimental Psychology*, *63*(10), 1953–1968.

McCabe, T. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*(4), 308–320.

NDepend. (2013, Jan). *NDepend Code Metrics Definitions.* http://www.ndepend.com/metrics.aspx.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*(5), 595–609.

between - functions

```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

def common(list1, list2):
    winners = []
    for item1 in list1:
        if item1 in list2:
            winners.append(item1)
    return winners

x = [2, 8, 7, 9, -5, 0, 2]
x_btwn = between(x, 2, 10)
print x_btwn

y = [1, -3, 10, 0, 8, 9, 1]
y_btwn = between(y, -2, 9)
print y_btwn

xy_common = common(x, y)
print xy_common
```

Output

```
[8, 7, 9]
[1, 0, 8, 1]
[8, 9, 0]
```

rectangle - class

```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def width(self):
        return self.x2 - self.x1

    def height(self):
        return self.y2 - self.y1

    def area(self):
        return self.width() * self.height()

rect1 = Rectangle(0, 0, 10, 10)
print rect1.area()

rect2 = Rectangle(5, 5, 10, 10)
print rect2.area()
```

Output

```
100
25
```

order - inorder

```
def f(x):
    return x + 4

def g(x):
    return x * 2

def h(x):
    return f(x) + g(x)

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

Output

```
5 2 7
```

whitespace - linedup

```
intercept = 1
slope     = 5

x_base  = 0
x_other = x_base + 1
x_end   = x_base + x_other + 1

y_base  = slope * x_base  + intercept
y_other = slope * x_other + intercept
y_end   = slope * x_end   + intercept

print x_base,  y_base
print x_other, y_other
print x_end,   y_end
```

Output

```
0 1
1 6
2 11
```

scope - samename

```
def add_1(added):
    added = added + 1

def twice(added):
    added = added * 2

added = 4
add_1(added)
twice(added)
add_1(added)
twice(added)
print added
```

Output

```
4
```

overload - plusmixed

```
a = 4
b = 3
print a + b

c = 7
d = 2
print c + d

e = "5"
f = "3"
print e + f
```

Output

```
7
9
53
```

initvar - good

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 0
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

Output

```
24
10
```

partition - unbalanced_pivot

```
pivot = 3
for i in [1, 2, 3, 4]:
    if (i < pivot):
        print i, "low"
    if (i > pivot):
        print i, "high"
```

Output

```
1 low
2 low
4 high
```

funcall - vars

```
def f(x):
    return x + 4

x = f(1)
y = f(0)
z = f(-1)
print x * y * z
```

Output

```
60
```

counting - twospaces

```
for i in [1, 2, 3, 4]:
    print "The count is", i


    print "Done counting"
```

Output

```
The count is 1
Done counting
The count is 2
Done counting
The count is 3
Done counting
The count is 4
Done counting
```

Table 1: Results by program version. (*) = regression reference, LOC = lines of code, CC = cyclomatic complexity, RT = reaction time. Main effects listed for version and experience. CE = prob. of common error, GR = grade, * = significance.

Type	Version	LOC	CC	Avg. Grade	Med. RT (s)	Effect (ver)	Effect (exp)
between	functions (*)	24	7	4.7	142.0		CE ↑ *
	inline	19	7	5.8	151.0		
counting	nospace (*)	3	2	8.8	55.0		
	twospaces	5	2	5.9	48.0	CE ↑ ***	
funcall	nospace (*)	4	2	9.1	29.0		
	space	4	2	8.8	27.0		
	vars	7	2	9.8	31.5		
initvar	bothbad (*)	9	3	8.7	59.0		
	good	9	3	7.1	56.0	GR ↓ *	
	onebad	9	3	7.7	56.0		
order	inorder (*)	14	4	8.7	50.0		
	shuffled	14	4	9.1	58.0	RT ↑ *	
overload	multmixed (*)	11	1	8.9	22.0		
	plusmixed	11	1	8.7	30.5	RT ↑ **	
	strings	11	1	8.5	35.0		
partition	balanced (*)	5	4	6.9	33.0		
	unbalanced	5	4	8.0	36.0	CE ↓ *	
	unbalanced_pivot	6	4	8.1	42.0		
rectangle	basic (*)	18	2	9.7	58.0		
	class	21	5	9.4	70.0		
	tuples	14	2	9.5	70.0	RT ↑ **	
scope	diffname (*)	12	3	7.2	41.0		CE ↓ *
	samename	12	3	6.7	44.5		
whitespace	linedup (*)	14	1	8.7	97.5		
	zigzag	14	1	8.5	100.0	CE ↑ *	