国立大学法人電気通信大学 / The University of Electro-Communications

Split-Paper Testing: A Novel Approach to Evaluate Programming Performance

著者（英）	Yasuichi Nakayama, Yasushi Kuno, Hiroyasu Kakuda
journal or publication title	Journal of Information Processing
volume	28
page range	733-743
year	2020-11
URL	http://id.nii.ac.jp/1438/00009808/

doi: 10.2197/ipsjjip.28.733

Journal of Information Processing Vol.28 733–743 (Nov. 2020)

[DOI: 10.2197/ipsjjip.28.733]

Regular Paper

Split-Paper Testing: A Novel Approach to Evaluate Programming Performance

Yasuichi Nakayama[1,a)] Yasushi Kuno[1,b)] Hiroyasu Kakuda[1,c)]

Received: November 14, 2018, Revised: June 26, 2019/March 23, 2020,
Accepted: July 28, 2020

Abstract: There is a great need to evaluate and/or test programming performance. For this purpose, two schemes have been used. Constructed response (CR) tests let the examinee write programs on a blank sheet (or with a computer keyboard). This scheme can evaluate the programming performance. However, it is difficult to apply in a large volume because skilled human graders are required (automatic evaluation is attempted but not widely used yet). Multiple choice (MC) tests let the examinee choose the correct answer from a list (often corresponding to the "hidden" portion of a complete program). This scheme can be used in a large volume with computer-based testing or mark-sense cards. However, many teachers and researchers are suspicious in that a good score does not necessarily mean the ability to write programs from scratch. We propose a third method, split-paper (SP) testing. Our scheme splits a correct program into each of its lines, shuffles the lines, adds "wrong answer" lines, and prepends them with choice symbols. The examinee answers by using a list of choice symbols corresponding to the correct program, which can be easily graded automatically by using computers. In particular, we propose the use of edit distance (Levenshtein distance) in the scoring scheme, which seems to have affinity with the SP scheme. The research question is whether SP tests scored by using an edit-distance-based scoring scheme measure programming performance as do CR tests. Therefore, we conducted an experiment by using college programming classes with 60 students to compare SP tests against CR tests. As a result, SP and CR test scores are correlated for multiple settings, and the results were statistically significant. Therefore, we might conclude that SP tests with automatic scoring using edit distance are useful tools for evaluating the programming performance.

Keywords: evaluating programming performance, constructed response tests, multiple choice tests, computer-based tests, split-paper tests

1. Introduction

Recently, the importance of programming education has been increasing rapidly worldwide; many pupils and students are starting to join programming classes day by day. The consequence is that there is a great need for evaluating the programming performance of these pupils and students, both for formative evaluation and grading.

Additionally, such evaluation will be a useful option for inclusion in college entrance examinations because computational thinking skills and programming skills will be useful (and even necessary) in many academic disciplines.

The problem is that there is no agreed upon method for evaluating programming performance that is both effective and practical. "Programming performance" here means the "ability to construct correct programs as solutions to solve clearly defined problems." Although practical programming tasks include upstream (requirements specification, design) and downstream (testing, maintenance) tasks, the ability to construct correct programs seems to be the starting point, and is an initial goal for introductory programming courses.

An obvious choice for evaluating the programming performance is to let the examinee write programs (on paper or by using a computer keyboard) from scratch. This method can be categorized as "constructed response" (CR) questions.

This method is effective in that, if examinees can write requested programs from scratch, it is a clear indication of the ability to write programs (at least at the level requested by the problem). However, this places a heavy burden on graders, who need to read and grade submitted programs. Automatic evaluation for such programs has been attempted, but is not in wide use at present (see the next section).

Alternative methods are various types of "multiple-choice" (MC) questions. Some show program code in a problem and present multiple choice questions about the code. Others are "fill-in-the-hole" type questions, in which several portions of program code are hidden (the holes), and the examinee chooses "correct" pieces for that hole from multiple choices.

A strong point of this method is its practicality; combined with a computer-based test (CBT) or with mark-sense cards, rating can be done automatically by using simple programs. Thus, a large volume of tests can be handled. However, many teachers and researchers are doubtful of its effectiveness, in that having the ability to answer fill-in-the-hole questions does not mean that one has the ability to write real programs.

In this paper, we propose yet another method for evaluat-

[1] University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[a)] nakayama@cs.uec.ac.jp
[b)] skuno@acm.org
[c)] kakuda@acm.org

© 2020 Information Processing Society of Japan

ing programming performance. Our method is a middle course among CR questions (programming from scratch) and MC questions (fill-in-the-hole), and it is both practical (judging is easily done by computer) and effective (high scores generally correspond to the ability to program from scratch).

Our method is called the "Split-Paper (SP) test," in which correct program code lines are separated line-by-line, removed of indentations and duplicates, mixed with similar but unneeded lines, and reordered to make a set of choice lines. The examinee chooses from those lines and orders them to construct an answer program. In our method, scoring is done automatically by using the edit distance [1] (Levenshtein distance) from correct answer programs. Our research question is whether the SP test, using an edit distance-based scoring scheme, measures the programming performance as in a CR test.

In the following sections, we describe the main idea of our method, its actual use in an examination, and the result of experimental tests we conducted. However, we first investigate the existing research on MC questions versus CR questions and automatic grading of programming tasks.

2. Related Work

2.1 MC Versus CR Questions

As explained above, MC questions and CR questions are well-known and have been around for a while. There has been a lot of research on their comparison. A report and a paper by Frederiksen [2], [3] discussed the important points of MC versus CR tests.

First, if MC tests were prepared in such a way that the examinee fully solves the stated problems in an ordinary way (as in the CR test) and then chooses from among multiple options, the MC and CR test formats would make no difference. However, if CR tests were to require examinees to come up with several hypotheses and construct answers on the basis of them, and MC tests were to present lists of hypotheses and let examinees choose from them, MC and CR test scores would be mostly unrelated.

Second, the ability to solve ill-structured problems, as described in Ref. [4], is an important outcome of higher education. However, problems stated in MC tests tend to be clearly stated (well-structured problems) and thus will not measure such ability. Free-format CR tests do not have such a limitation.

These two findings seem to suggest that MC and CR tests might measure the same thing in some cases when thoughtfully prepared, which is not always easy.

Another paper by Simkin and Kuechler [5] contains an extensive survey of related work and is well written, and it is mainly targeted at the domain of programming performance evaluation. We briefly introduce their discussion on MC versus CR tests, along with supplementary comments.

First, some of the existing researches [6], [7], [8] claim that MC and CR questions could largely measure the same thing. However, in Ref. [5], it is pointed out that those pieces of research first define skills to be measured with MC questions, and then show that CR questions are not needed to measure those skills. Therefore, in such research, skills that cannot be measured with MC questions are not investigated.

Table 1 Summary of Bloom's Taxonomy.

Level	Description	Evidence
1. Knowledge	Rote memory	Answer T/F or MC questions
2. Comprehension, translation, interpretation, extrapolation	Assimilation into learner's frame of reference, give meaning, change representation	Understand similar programs, put in own words, classify material, predict consequences
3. Application	Abstraction toward new situation	Use learned techniques and knowledge to new situation
4. Analysis	Decomposition and understanding relationships	Recognize unstated assumption
5. Synthesis	Combine learned elements	Knowledge creation, fill gaps in existing knowledge
6. Evaluation	Makes judgement about value of learned information	Judge directions of knowledge acquisition

We have frequently observed that those who can score well on MC questions cannot write anything when asked to write program on a blank sheet. This "start from scratch" skill might be one that cannot be measured with MC questions.

Second, in some of the existing researches [9], [10], [11], the researchers do not categorize carefully what their MC questions measure; they regard a set of MC questions as a uniform body — such an assumption is unlikely to hold, of course. On the contrary, the authors of Ref. [5] use Bloom's Taxonomy [12] (see **Table 1**) to categorize various skills required for programming. They claim that levels 1 through 3 apply to introductory programming classes, and show that they can construct MC questions for all of the three levels.

Our observation on existing MC tests on programming is that they are often targeted at the lower levels. Even questions aimed at a higher level can often be answered by memorizing frequent patterns in representative programs.

Third, in Ref. [5], it is stated that creating MC problems targeted at level 3 is possible but "surprisingly difficult." The statement matches well with our observation noted above.

Given that MC questions that evaluate the programming performance at all three levels are possible but in fact very hard, other test schemes that can evaluate all levels and that are also easy to use will be valuable; that is where our proposed method comes in.

2.2 Automatic Grading of Programming Tasks

In the previous section, we stated the problem that CR tests for programming tasks are desirable but pose much of a burden on grading. However, the problem vanishes in thin air if automatic grading with computer programs is possible. In fact, there have been many such attempts [13], [14] yet they are currently not widely used (as noted previously). We will examine the causes here.

Some automatic graders are based on online judgement systems built for programming contests such as ACM ICPC; in Ref. [15], the use of such systems in automatic grading for programming classes is discussed. These graders apply many pieces of test data toward submitted programs, and a binary result is returned — "correct" or "not correct." This kind of behavior is apparently not satisfactory for assessment in programming classes,

© 2020 Information Processing Society of Japan

because we would like to score "totally wrong" and "near miss" differently. In fact, in Ref. [15], the use of additional manual grading for programming styles is discussed. Alternatively, as described in Ref. [16], automatic graders may rescue some near-miss problems. However, the repertoire of such rescues is limited, and an additional effort is required in describing extra patterns or data for it.

As a radically different approach, in Ref. [17], a rubric for grading computer programs is first defined, and a machine-learning-based grader is built for grading programs against the rubric. The grader examines several features in input programs and uses a learned weight on those features for grading.

The weak point is that the judgement of the grader (pass/not pass) differed from human judgement in approximately 20% of the cases. As this kind of grader is not based on the correctness of the program but on the (non-)existence of features in the code, this difference is unavoidable — such a system will be nice for providing advice, but not for grading in a class.

Another point is the kind of programming tasks. Many of the automatic graders are used for home assignment. In this case, students invest many hours on an assignment, so the size and complexity of programs are relatively large. Of course the use of an automatic grader here is alright because students can obtain instant feedback, allowing for a lot of trial-and-error.

However, as was pointed out in Ref. [18], for formative and summative evaluation, small-scale lab-testing within an hour or so will be more appropriate. We should use concise and accurate problem specifications with clear input/output specification. In this case, a solution other than writing programs in a free-format might be viable; this is the point where our idea comes in.

2.3 Parsons Problems/Mangled Code

There are several pieces of research on program-line reordering tests. Parsons et al. [19] proposed such a test, named "Parsons Programming Puzzle." Their intent was to use such quizzes as a tool in programming education and not as an evaluation scheme.

Denny et al. [20] interviewed students in college CS1 (introduction to programming) classes on how they feel about Parsons-like problems. From the result, they concluded that the tests, which have completely shuffled lines, are too difficult, and they chose to use a variant in which each "correct" line is paired with a corresponding "incorrect" line and each pair is placed adjacently.

Then, they used both their Parsons variant and CR test for end-term exam with 13 students. To hand-score all problems, they designed detailed marking rubrics for all questions. From the statistics test for the Parsons variant and CR scores, $\rho = 0.53$ was obtained. Although the correlation factor is moderately high, they used hand scoring with detailed mark rubrics, so the cost of scoring was high.

Additionally, their scheme is rather different from our SP tests in that each "correct" line is paired with a corresponding "wrong" line placed adjacently. We suspect that providing explicit correct-wrong pairs might make the test rather close to the MC test.

Ericson et al. [21] compared this Parsons variant with correct-wrong pair (same as Ref. [20]) against code writing in classroom settings, and reported that the Parsons variant took significantly less time. The result also suggests that the correct-wrong pair variant requires skills different from CR tests.

Cheng et al. [22] use the term "mangled code" to denote their version of a Parsons-like problem. They compared mangled code tests with CR tests for a CS1 end-term exam with 473 students. Their mangled code is created by simply splitting the correct program line-by-line and shuffling it (duplicate lines are left intact, and no incorrect lines are added).

They used two programming problems. Every student solved one as a mangled code test and the other as a traditional CR test; the assignment was done at random. Scoring is done manually (by teaching assistants).

They report that the scoring times for the two types of problems were not different for one problem (20 min. for mangled, 22 min. for CR) and different for the other (9 min. for mangled, 16 min. for CR); these times were the average time for scoring 50 answers.

As a result, Spearman's rank correlation of $\rho = 0.65$ between the mangled test score and CR test score was obtained. Although the correlation factor was rather high, we suppose that their mangled code test is again close to MC tests because no incorrect lines are included in the choices and duplicates are not removed. Also, note that scoring by hand requires a high cost for mangled code test (they report 50% labor compared with CR tests at best).

3. Split-Paper Method

3.1 The Basic Idea

In 2012, the authors, together with other colleagues, formed the "Joho Nyushi (Informatics Entrance-Examination in Japanese) Study Group" (JNSG), whose purpose was to investigate an effective, useful, and practical scheme for college entrance examinations on the subject of informatics in Japan.

One of our goals was to develop a useful and practical method for evaluating programming performance. As a solution, we developed a scheme called the "split-paper" method (or TANZAKU method in Japanese).

The idea is simple; in the era of punched cards, we punch each line of a program on a paper card, and a deck of paper cards represents a single program. Should we mistakenly drop the deck on the floor, gathering the cards "correctly" would surely require programming ability!

In the actual test scheme, we do not use paper cards of course. We split the "correct answer" program into each line, add some "wrong answer" lines, reorder (or shuffle) them, and prepend them with choice symbols. On the paper test, we request examinees to list symbols (corresponding to the answer program) in the answer field. In the case of CBT, a dragging interface can be incorporated, with which examinees can directly construct an answer program on the screen.

The name "split-paper" came from the image of writing each line on a strip of paper and reordering the strips as necessary.

3.2 An Example: Problem for JNSG 2013 Tests

In **Fig. 1**, we present the problem we used for the JNSG's first nation-wide college entrance examination simulated tests held on May 18, 2013 [23]. The intention of the test is to examine the

© 2020 Information Processing Society of Japan

> Read the following sentences and answer problems 1 and 2.
> The following program does the following: "when positive integer n is entered, print out the numbers from 1 to n."
> input an integer to n.
> repeatedly change i from 1 to n by 1.
> print out i.
> end of "repeat."
> If the above program is represented with the choices below, the answer will be: AGLH.
> **Choices**
> A. input an integer to n.
> B. $j \leftarrow 0$.
> C. $j \leftarrow i$.
> D. $j \leftarrow j + 1$.
> E. $j \leftarrow j + i$.
> F. $j \leftarrow -j$.
> G. repeatedly change i from 1 to n by 1.
> H. end of "repeat."
> I. if i is an even number,
> J. if i is not an even number,
> K. end of "if."
> L. print out i.
> M. print out j.
> Create a program that does the following. Use as many choice lines as you like. You can use the same line multiple times.
> **Problem 1.** When positive integer n is entered, print out a total of odd integers not exceeding n (for example, when 7 is entered, the program will print out 16 because $1 + 3 + 5 + 7 = 16$).
> **Problem 2.** When positive integer n is entered, print out integers from 1 to n, with the even numbers' sign negated (for example, when 7 is entered, the program will print out $1, -2, 3, -4, 5, -6, 7$).

Fig. 1 Problem for JNSG 2013 tests.

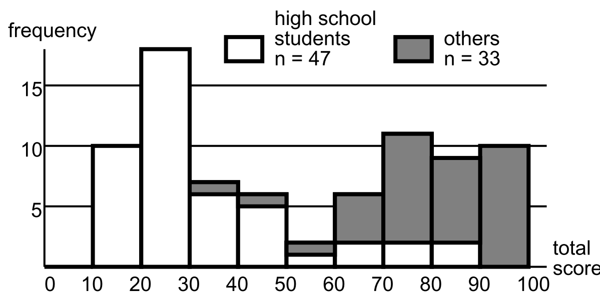


Fig. 2 Total score distribution of JNSG 2013 test.

ability to program simple loops, conditions, and sequential execution with variable assignments.

JNSG tests were also held in February 2014, 2015, and 2016, all with split-paper method programming problems. We have chosen the first one here because it is in the most basic form (later tests include additional subproblems for those who have not learned programming).

JNSG 2013 Tests were held in 5 cities with a total of 41 examinees. Additionally, several high school teachers cooperated with us and held the same test at their high schools with a total of 39 examinees. Among the 80 examinees, the number of high school students was 47. The others were high school teachers, college students, or researchers who have various interests. The distribution of the total score is shown in **Fig. 2**. As can be seen, high school students and the other examinees belonged to different groups.

Within the full score of 100 points, the split-paper programming test of Fig. 1 occupied 15 points (problem 1: 7 points, problem 2: 8 points). Correct answers and a scoring guide are shown in **Fig. 3**.

Figure 4 shows a scatter plot of the split-paper (SP) programming score against the total score for the JNSG 2013 test. Clearly, most of the high school students had not learned programming and thus could not program, and many of the "other" peoples could program.

> Problem 1: ABGJEKHM, BAGJEKHM
> Problem 2: AGCIFKMH, AGCFJFKMH
> If meaningless lines are included, subtract 1 point for each. If the program does not produce the correct output, subtract 3 points for each lacking statement or extra statement.

Fig. 3 Answers and scoring guide.

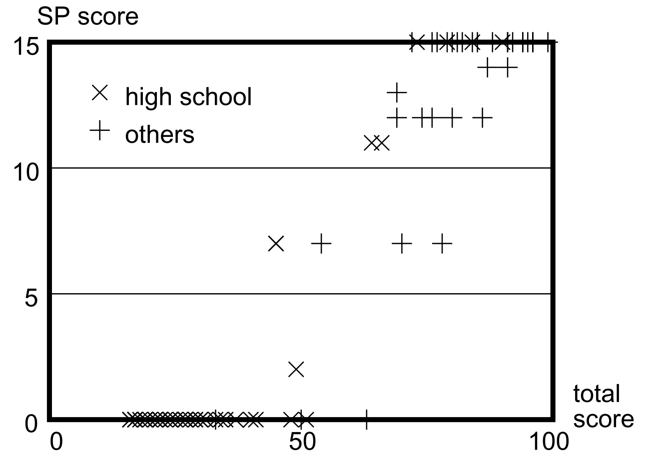


Fig. 4 Scatter plot of SP score against total score in JNSG 2013 test.

The result suggests that split-paper tests for programming can be used to judge whether an examinee can actually program or not. More accurately, a good score on split-paper tests and on a "write-on-a-blank-sheet" programming performance evaluation might be highly correlated, which was our initial research question. Although early JNSG tests used hand scoring, we later come up with the idea of edit distance-based scoring, and accumulated some experiences; now we always use SP tests combined with edit distance-based scoring.

4. Evaluation Within College Programming Classes

In the JNSG Tests we could discriminate those who can program from those who cannot. However, our goal was to use our method for a more detailed evaluation (as in a programming class). For this purpose, we should compare our method against CR (write-on-a-blank-sheet) programming tests.

Therefore, we experimented with our test scheme in college programming classes. The experiments were conducted for the University of Electro-Communications (UEC) programming class "Programming" on December 19, 2016 and January 16, 2017.

The subjects belonged to a second-grade programming class, held over 15 weeks (90 min. for each week) with both lecture and practice. C language was used, and topics on various algorithms and data structures were included.

Our experiments (in the 10th and 11th week) were conducted as formative evaluation tests within the classes. Students were requested to solve two problems in a series, each in 10 minutes. Those problems were called problems C and D for the 10th week and problems E and F for the 11th week (see Appendix).

© 2020 Information Processing Society of Japan

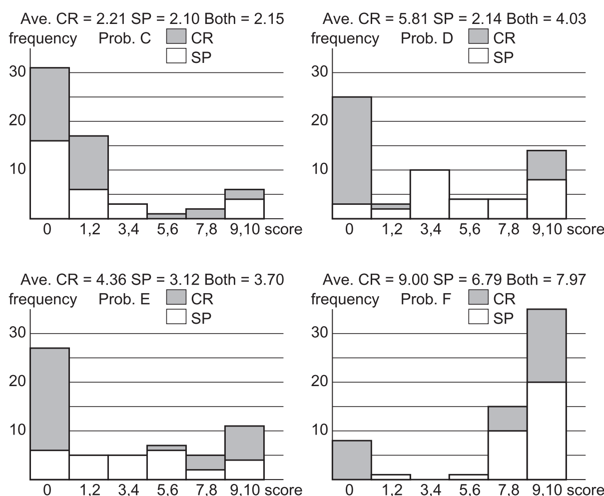


Fig. 5 Histogram of test scores.

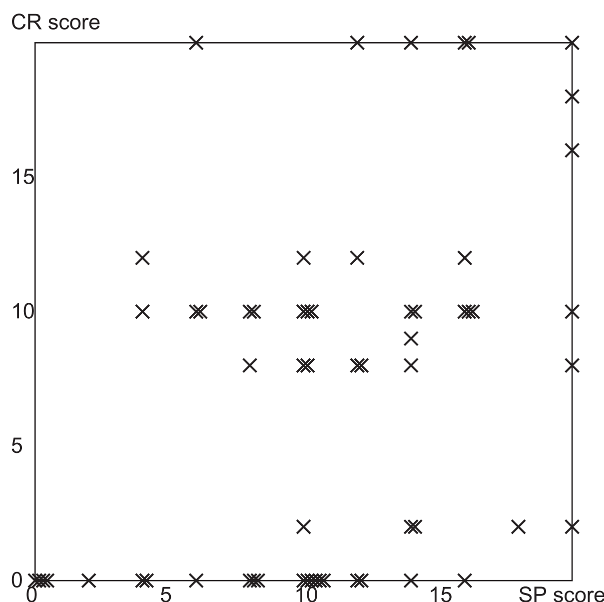


Fig. 6 Scatter plot of SP vs. CR score.

Table 2 Category count for each pair with average score difference.

category	norm.	rev.	dif. SP	dif. CR	same
count	870	396	314	151	39
percentage	49.2%	22.4%	17.7%	8.5%	2.2%
ave. dif. SP	7.3	5.5	6.2	–	–
ave. dif. CR	9.8	3.9	–	8.3	–

The number of students registered to the subject was 98. Among them, 60 students had attended both weeks 10 and 11, so we used the scores of those 60 students in the following analysis.

All of the problems were prepared in both split-paper (SP) and CR (write on a blank sheet) formats. We assigned symbols C1, D1, E1, and F1 for the SP format and C2, D2, E2, and F2 for the CR format. Students were randomly assigned to one of four groups (the groups differed between the 10th and 11th week) and solved problems in C1-D2, C2-D1, D1-C2, D2-C1 ordering for the 10th week and E1-F2, E2-F1, F1-E2, F2-E1 ordering for the 11th week.

For the CR test, rating was done manually by the lecturer (one of the authors), with a full score being 10 points each. For the SP test, rating was done with a program, the score being $max(0, 10-2d)$, where d was the edit distance between the correct program and the answer. A more detailed discussion on rating is given in the next chapter.

In **Fig. 5**, we show histograms for each of the problems, along with the average score for CR, SP and both. As the result, the average score was the highest for problem F, followed by problems D, E, and C in decreasing order.

In **Fig. 6**, we show a scatter plot of CR scores against SP scores [*1]. From this plot, the SP and CR scores seem to be correlated. We applied a non-parametric statistical test with Spearman's rank-order correlation. The result was $\rho = 0.407$, $p < 0.0012$. Therefore, we can say that the SP test and CR test scores in our experiment were weakly correlated and that the result was statistically significant.

The value of $\rho = 0.407$ was not large. As shown in Fig. 5, the average score for corresponding problems used in cross-comparison (C vs. D and E vs. F) differed, so this might be the cause.

Our interest is whether it is appropriate to use the SP score for evaluating the programming performance (in place of the CR score). It would be appropriate if an examinee who scored higher for CR also scored higher for SP (normal ordering) and not if he scored higher for CR and scored lower for SP (reverse ordering). With 60 students, there were 1770 (= $\frac{60\times59}{2}$) pairs. Each pair of students, whose SP and CR scores sp_1, cr_1 and sp_2, cr_2 respectively, could be categorized as in either of the following.

- normal order — $sp_1 > sp_2$ and $cr_1 > cr_2$, or $sp_1 < sp_2$ and $cr_1 < cr_2$.
- reverse order — $sp_1 > sp_2$ and $cr_1 < cr_2$, or $sp_1 < sp_2$ and $cr_1 > cr_2$.
- differ for SP — $sp_1 \neq sp_2$ and $cr_1 = cr_2$.
- differ for CR — $cr_1 \neq cr_2$ and $sp_1 = sp_2$.
- same for both — $sp_1 = sp_2$ and $cr_1 = cr_2$.

We have counted the number corresponding to each of the above categories. Additionally, we would like to examine score differences among each pair, so we calculated the SP score difference and the CR score difference for each pair, and took the average within each of the above categories. **Table 2** summarizes the results.

As the result, (1) the number of "reverse ordering" case was approximately 22% of all the pairs, (2) the number of "normal ordering" cases was more than twice that of "reverse ordering" case, and (3) the average score difference is larger in "normal ordering" cases compared with the "reverse ordering" cases.

Finally, the class had an end-of-term exam on February 20th, and one of the exam problems was a CR programming problem (write a program on a blank sheet). All the 60 students took the exam.

The exam contained several other problems, and took 90 min. as a whole. Scoring was done manually by one of the authors; the maximum score was 32 (full score is 40) and the minimum score was 0, whose histogram is shown in **Fig. 7**.

[*1] Note that each student took two CR tests and two SP tests. Therefore, the maximum values of the CR and SP scores were both 20 points for each student.

© 2020 Information Processing Society of Japan

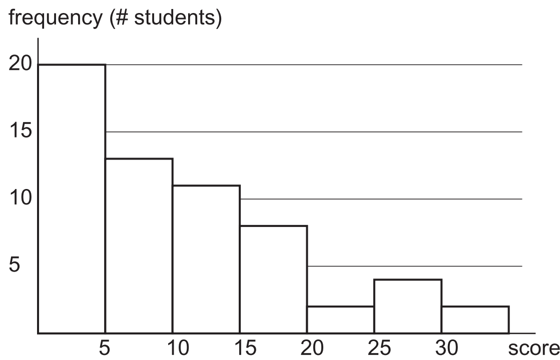


Fig. 7 Histogram of CR (Exam.) score.

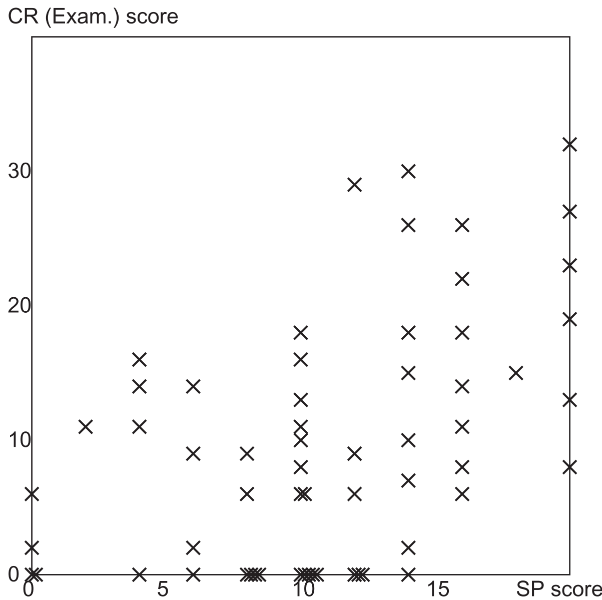


Fig. 8 Scatter plot of SP-CR (Exam.) score.

Table 3 Category count for each pair with average score difference (Exam).

category	norm.	rev.	dif. SP	dif. CR	same
count	990	434	156	169	21
percentage	55.9%	24.5%	8.8%	9.5%	1.2%
ave. dif. SP	7.6	5.6	5.4	–	–
ave. dif. CR	6.5	2.0	–	5.0	–

From the histogram, we can say that the exam problem was not easy for the students because the number of students monotonically decreased as the score increased. However, as $\frac{2}{3}$ of students earned more than or equal to 5 points, it is not the case that the score was totally meaningless.

We have taken the SP score from our experiments and plotted it against the exam CR score, result being **Fig. 8**. Now the SP and exam CR score seems to be more correlated.

We again applied Spearman's rank-order correlation. The result was $\rho = 0.478$, $p < 0.00011$. Therefore, we might say that the SP and exam CR scores were weakly correlated, and the result was statistically significant.

In **Table 3**, we show category count using the SP score in our experiment and the exam CR score. We multiplied the exam CR score by 0.5 because the full score (40) was twice that of the experiment. Compared with the previous experiment, the count of "differ SP" category significantly decreased.

However, the result was generally similar to the experiment in that (1) the number of "reverse ordering" cases was approximately 25% of all the pairs, (2) the number of "normal ordering" cases was more than twice that of the "reverse ordering" cases, and (3) the average score difference was larger in the "normal ordering" cases compared with the "reverse ordering" cases.

```
function dist(s, t)  // distance of s_1 ··· s_m and t_1 ··· t_n
  for i in 0..m do a[i][0] := i;   // initialize the leftmost column
  for j in 0..n do a[0][j] := j;   // initialize the topmost row
  for i in 1..m do
    for j in 1..n do begin   // for all remaining cells
      m := (if s_i = t_j then a[i-1][j-1] else a[i-1][j-1]+1);
      if i > 1 ∧ j > 1 ∧ s_{i-1} = t_j ∧ s_i = t_{j-1} then
          m := min(m, a[i-2][j-2]+1);
      m := min(m, a[i-1][j]+1);
      m := min(m, a[i][j-1]+1);
      a[i][j] := m
    end;
  return a[m][n]   // final result
```

Fig. 9 Edit distance algorithm used for scoring.

5. Scoring Scheme for SP Tests

5.1 Consideration on SP Tests Scoring

The largest benefit of SP testing is the applicability of automatic scoring, so how to do this is an important topic. It is not difficult to exhaust all "correct programs" (programs that work as expected) from choice lists in advance, when expected programs are not that long (perhaps up to 20 lines or so).

There is always some arbitrariness of line orderings in program code (ex: declarations `int i;`, `int j;` and `int k;` can be written in any order), leading to an exponential explosion. For such cases, we might pose additional restrictions in the problems ("variable declarations must be placed in alphabetical order of their names") or use combined choice line (as in `int i, j, k;`).

The problem is that "correct or not" is too coarse as a rating scheme. We all know that a totally wrong program and a mostly correct program with minor mistakes are of quite different value. Human graders can naturally account for such differences, so we would like to do the same for automatic rating for our SP tests.

5.2 Edit Distance Algorithm

We have been using edit distance (Levenshtein distance) to assign a partial score on JNSG tests since year 2014. The edit distance of two sequences, *A* and *B*, is the minimum number of editing operations (insertion of a line, deletion of a line, swapping of an adjacent pair of lines, or modification of a line) that will transform *A* into *B*.

Figure 9 shows the actual algorithm, which computes the edit distance among two strings *s* and *t* using DP (dynamic programming). We used Pascal-like notation for readability (actual code is written in Ruby and Awk).

An array element $a[i][j]$ holds the edit distance among two substrings $s_1 \cdots s_i$ and $t_1 \cdots t_j$, or $\mathrm{dist}(s_1 \cdots s_i, t_1 \cdots t_j)$. The first two lines of the function state that substring $s_1 \cdots s_i$ and $t_1 \cdots t_j$ can be converted to null string by deleting *i* and *j* characters respectively (*i* or *j* edit operation(s)).

All of the other array elements are filled in the inner loop body.

© 2020 Information Processing Society of Japan

Journal of Information Processing Vol.28 733–743 (Nov. 2020)

pt.	explanations
10	correct program
9	really small syntax mistake
8	one minor mistake
6	two minor mistakes
2	many lines of code with many errors
0	otherwise

Fig. 10 CR scoring scheme for experiment.

q2: remove the top element from a circular list `L` containing integers

- correct program — 15 pts.
- runs only when `L==NULL` or `L->next != L` — 10 pts.
- runs only when `L==NULL` or `L->next == L` — 10 pts.
- runs only when `L==NULL` — 5 pts.

(for all above, subtract 2 pts each for minor mistakes and 1pt for really minor mistake; score serious mistakes with 0 pts.)

q3: partition a circular list `L` into three using the head element as the pivot

- correct program — 15 pts.
- runs only when `L==NULL` or `L->next==L` — 8 pts.
- runs only when `L==NULL` — 3 pts.

(for all above, subtract 2 pts each for minor mistakes; score a serious mistake with 0 pts.)

q4: quick sort a circular list using the subroutines of q2 and q3

- correct program — 10 pts.
- runs only when `L==NULL` — 3 pts.

(for all above, subtract 2 pts each for minor mistakes; scores a serious mistake with 0 pts.)

(full score 40 pts)

Fig. 11 CR scoring scheme for end-term exam.

The first line of the body states that dist($s_1 \cdots s_i$, $t_1 \cdots t_j$) can be the same as dist($s_1 \cdots s_{i-1}$, $t_1 \cdots t_{j-1}$) if s_i is equal to t_j (because no modification is necessary), or larger by one otherwise (s_i have to be changed to t_j). This is the tentative value m. However, when the lengths of both substrings are larger than 1 and $s_i = t_{j-1}$ and $s_{i-1} = t_j$ and additionally dist($s_1 \cdots s_{i-2}$, $t_1 \cdots t_{j-2}$) + 1 < m, then m can be set to this smaller value because single swapping of two characters at the tail will convert $s_1 \cdots s_i$ to $t_1 \cdots t_j$. Finally, the removal of the last character of $s_1 \cdots s_i$ or $t_1 \cdots t_j$ is considered, and the final (smallest) m value is recorded in a[i][j]. When the loops are finished, a[m][n] holds the edit distance value among s and t.

For SP test scoring, if an examinee's answer string is s, and there are multiple "correct" answers $t_1 \cdots t_n$ for the problem, we compute the minimum edit distance d = min(dist(s,t_1), $\cdots$, dist(s,t_n)). Note that if d is 0, s is identical to one of the correct answers.

5.3 Scoring Scheme Used in the Experiment

Of course, the importance of each line in a program varies, so the same edit distance from (one of) the correct program(s) does not necessarily mean the same "wrongness." However, in our experience with the JNSG test, it works fairly well.

Therefore, in our experiment we compared the edit distance-based scoring of SP tests against manually scored CR tests. As described above, the SP score is calculated as $max(0, 10 - 2d)$, where d is the minimum edit distance among answer sequences and sequences corresponding to one of the correct programs [*2].

Figures 10 and **11** show the CR scoring scheme for the experiment problems (problems C, D, E, and F) and end-term exam problems (writing three successive C functions, 40 pts. in total).

From the experiences of the authors (all of the authors had taught programming more than 30 years and have experiences on scoring CR programming problems accordingly), the scoring scheme of Figs. 10 and 11 seems ordinary. However, it looks rather complex compared with the simple edit-distance-based scoring scheme used in our SP tests.

[*2] Note that there might be several "correct" programs for many problems, so taking the minimum is required.

6. Discussions

6.1 Usefulness of SP Testing

In the Introduction section, we noted that there is a large volume of evaluation of the programming performance. The situation is especially true in Japan.

Japan's Ministry of Education, Culture, Sports, Science and Technology (MEXT) has published new curriculum guidelines for primary and upper secondary schools in March 2017, which include programming education in elementary schools for the first time in Japan's history.

Although the elementary school curriculum is aimed at programming experience (perhaps with graphics-based languages), the effect is that more text-based programming will be taught in junior-high and high schools, leading to a large volume of performance evaluation tasks. It also might be possible that programming problems will be included in the college entrance examination test in the near future.

All of these require an effective testing scheme for programming performance. MEXT is planning to incorporate CBT for Japan's common college entrance examination test, so evaluating programming performance with CBT will be required. Although MC tests have been widely used with CBT, they are not suitable for the evaluation, as discussed previously. While CR tests (write on a blank sheet) will be an appropriate method (or "standard" method at least), automatic scoring is an unsolved problem. Therefore, we think that SP tests might be a viable alternative that satisfies our social needs.

6.2 Evaluating Programming Performance

Although we have repeatedly used the term "programming performance," this broad term is quite difficult to define.

However, we are considering educational settings, so we exclude such things as the performance of professional programmers in large and complex projects. Moreover, school programming projects which may last weeks are also excluded; the outcome of such projects will accompany written reports or human presentations, so human graders will be needed anyway.

Our target is in-class or end-of-the-term tests, which are mandatory in schools and pose a heavy burden on graders. Therefore, the base line is traditional CR (write-on-a-sheet) programming tests and the ability to write programs (at most 30–50 lines, at most an hour or so) on an answering sheet.

Therefore, our research question is whether SP test scored by using the edit-distance-based scoring scheme measures the programming performance as do CR tests. We used traditional CR tests as the baseline because they are in wide use; the appropriateness of CR tests themselves will be another interesting research topic.

© 2020 Information Processing Society of Japan

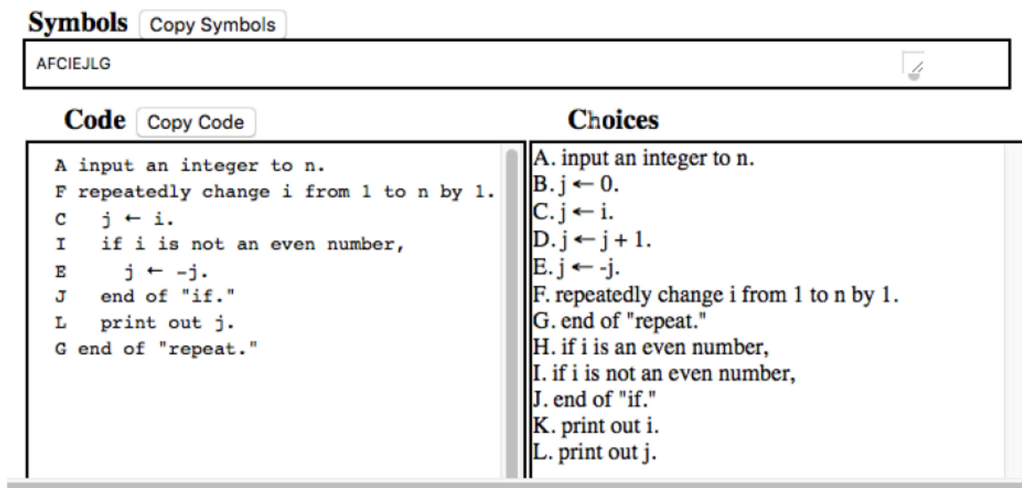


Fig. 12 Drag-and-drop interface.

6.3 Appropriateness of SP Testing

For tests of any domain, accuracy (closeness of the test score against the ability of the examinee) largely depends on how to construct problems. Even in the traditional CR (write-on-a-sheet) programming tests, a test will measure nothing if the problems are too easy or too difficult.

With respect to the difficulty of the SP test, we think we might be able to use a policy of "the same problem as on the CR test." For example, Fig. 5 shows the score distribution of CR and SP tests for the same problem. In three problems out of four, both scores of 0 and 9–10 (full score) have a frequency of at least 2 for both of the CR and SP formats, which suggests the appropriateness of their difficulty.

An exception is problem F, for which there were no students with a score of 0 for SP. However for problem F on the CR test, the frequency of full scores was the highest, suggesting that this problem was too easy for a traditional CR test, which led to the SP test being too easy.

As CR programming tests are widely used with an appropriate level of problem difficulty, we suppose that we can apply those difficulty-controlling skills for tuning SP tests also. Of course, this is only a guess, and we need to do more research, but there is a possibility that such a policy would work well.

Then, how about the appropriateness of using SP tests in place of CR tests? Stated differently, do SP tests scored by using edit-distance-based scoring scheme measure the programming performance as do CR tests (this is the research question for the paper)?

In this respect, we do not have a concrete answer yet. However, we have shown that (1) SP and CR test scores for our experiment (including end-term exams) were weakly correlated and that the result was statistically significant, and (2) reverse ordering (a case where ordering of SP and CR scores was reversed) was within 25% of all student-student pairs.

Therefore, we think it might be possible to use the SP test score in place of the CR test score, for which we should investigate further of course.

6.4 Tool Support

As noted previously, CBT for the split-paper test will be beneficial in that the drag-and-drop interface can be used to directly construct programs on the screen.

We developed such a tool (**Fig. 12**). This SP CBT tool is a Ruby script that accepts a problem description (problem statement and list of choices) and generates an HTML file equipped with JavaScript code. When the resulting HTML file is displayed in a standard Web browser, the dragging interface is activated and the user can use drag-and-drop to construct one's own answer program.

There are buttons labeled "Copy Symbols" and "Copy Code" on the screen. The former button copies a string of choice symbols to a clipboard, and the user can paste the symbols in the answer field. The latter button copies program code (without choice symbols) to a clipboard, and the user can paste the code with a text editor to issue a test run (if the problem uses a real programming language).

We have been using this SP CBT tool in UEC's 1st-grade (800 students) computer literacy and introductory programming classes since April 2017. Both classes use Moodle CMS, and the SP CBT tool-generated HTML is embedded within an ordinary Moodle test page. SP tests are used both for in-class exercises and for the end-of-the-term examination. Although some of the students are a little confused when they first see the interface, they soon get accustomed to it.

There we have noticed that paper-based testing (as in our experiment described above) and tests using the drag-and-drop interface have different characteristics, because examinees can view resulting (complete) programs on the screen in the latter. Of course, examinees can write down their programs in the margins of a test sheet (or blank sheet that is supplied), but it takes time to modify written programs on paper, so trial-and-error is more difficult. We would like to investigate this difference in experiments.

7. Summary and Future Directions

We have been using split-paper (SP) testing that use edit-distance-based scoring for several years for JNSG tests and felt that SP testing is both an effective and practical method for evaluating the programming performance. However, we had not had concrete data to support our intuition.

© 2020 Information Processing Society of Japan

In this paper, we reported our experiments comparing SP and CR testing in college programming classes. The CR testing scheme used was "write-on-a-blank-sheet," which is the standard, most widely-used method for evaluating the programming performance in classroom settings.

We have used a balanced experimental design in which each of the problems was solved in the SP format for half of the students and the CR format for the other. The class also had a CR style end-of-term exam, and the score was also available.

As a result, the students' SP and CR scores were weakly correlated in a (1) within-experiment SP versus CR score analysis and also in a (2) within-experiment SP score versus end-of-term CR score analysis; both of the results were statistically significant. Therefore, we conclude that the split-paper style of testing is possibly effective — it can be an appropriate method for evaluating the programming performance.

Also note that the elimination of costly manual scoring would be a huge benefit for many classroom settings; SP tests can be useful both in formative evaluation during class and end-term summative evaluation.

Research on SP testing over programming tasks has only just begun. While we only analyzed total test scores among two formats, programming skills apparently consist of many orthogonal dimensions. Examples are problem understanding, algorithm design, data structure comprehension and design, procedure-level program structuring, and the application of small-level coding idioms. We should compare SP and CR tests in an evaluation for each of those dimensions.

Automatic rating is another important topic. We are currently using edit distance (Levenshtein distance) [1] from the correct program, which seems robust and appropriate. However, the nature and appropriateness of this scheme should further be investigated.

The use of CBT with SP testing is undoubtedly important. We already mentioned the merit of "viewing a complete program on the screen" in SP CBT testing. We would like to investigate the volume and extent of this merit.

Our SP CBT tool is in its early stages and needs further tuning and enhancement. Many freshmen students at UEC are using our tool with no problem. However, usability testing and UI tuning could enhance its usefulness further.

Another possibility is using the "SP exercise tool" — just another name for the SP CBT tool. The tool could be used to support learners' in-class and/or home exercises. The tool can provide useful cue for constructing exercise programs (because code snippets are included as choice lines), and it can also be enhanced to provide additional advice when the learner makes mistakes.

In summary, SP testing is a general framework for examining the programming performance (or programming activity itself), and has good affinity with tool support. We would like to investigate its possibilities further.

References

[1] Gusfield, D.: *Algorithms on String, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press (1997).

[2] Frederiksen, N.: The real test bias, *ETS Resarch Report*, Vol.1981, No.2, Educational Testing Service (1981).

[3] Frederiksen, N.: The real test bias: Influence of testing on teaching and learning, *American Psycologist*, Vol.39, No.3, pp.193–202 (1984).

[4] Simon, H.A.: The Structure of Ill Structured Problems, *Artificial Intelligence*, Vol.4, pp.181–201 (1973).

[5] Simskin, M.G. and Kuechler, W.L.: Multiple-Choice Tests and Student Understanding: What Is the Connection?, *Decision Science Journal of Innovative Education*, Vol.3, No.1, DOI: 10.1111/j.1540-4609.2005.00053.x (2004).

[6] Bennett, R.E., Rock, D.A. and Wang, M.: Equivalence of Free-REsponse and Muitliple-Choice Items, *Journal of Educational Measurement*, Vol.28, pp.77–92, DOI: 10.1111/j.1745-3984.1991.tb00345.x (1991).

[7] Traub, R.E.: On the equivalence of the traits assessed by multiple-choice and constructed-response tests, Bennett, R.E. and Ward, W.C. (Eds.), *Construction Versus Choice in Cognitive Measurement*, pp.29–44, Routledge, ISBN 978-0805809640 (1993).

[8] Wainer, H. and Thissen, D.: Combining Multiple-Choice and Constructed Response Test Scores: Toward a Marxist Theory of Test Construction, ETS Research Report No.92-23, Educational Testing Service (1992).

[9] Bacon, D.R.: Assessing Learning Outcomes: A Comparison of Multiple-Choice and Short-Answer Questions in a Marketing Context, *Journal of Marketing Education*, Vol.25, No.1, pp.31–36 (2003).

[10] Kuechler, W.L. and Simkin, M.G.: How Well Do Multiple Choice Tests Evaluate Student Understanding in Computer Programming Classes?, *Journal of Information Systems Education*, Vol.14, No.4, pp.389–399 (2003).

[11] Walstad, W. and Becker, W.: Archievement Difference of Multiple-Choice and Essay Tests in Economics, CBA Faculty Publications 34, University of Nebraska – Lincoln (1994).

[12] Bloom, B., Englehart, M., Furst, E., Hill, W. and Krathwohl, D.: A Taxonomy of Educational Objectives, Handbook I: Cognitive Domain, David McKay Company (1956).

[13] Douce, C.: Automatic Test-Based Assessment of Programming: A Review, *ACM Journal of Educational Resources in Computing*, Vol.5, No.3, Article 4 (Sep. 2005).

[14] Lhantola, P., Apheniemi, T., Karavirta, V. and Seppälä, O.: Review of Recent Systems for Automatic Assessment of Programming Assignments, *Proc. 10th Koli Calling International Conference on Computer Education Research*, pp.86–93 (Oct. 2010).

[15] Cheang, B., Kurnia, A., Lim, A. and Oon, W.-C.: On automated grading of programming assignments in an academic institution, *Computers & Education*, Vol.41, No.2, pp.121–131 (2003).

[16] Morrs, D.S.: Automatic Grading of Students' Programming Assignments: an Interactive Process and Suite of Programs, *33rd ASEE/IEEE Frontiers in Education Conference*, DOI: 10.1109/FIE.2003.1265998 (2003).

[17] Srikant, S. and Aggarwal, V.: A System to Grade Computer Programming Skills using Machine Learning, *Proc. KDD'14*, pp.1887–1896 (2014).

[18] Daly, C. and Waldron, J.: Assessing the Assessment of Programming Ability, *Proc. SIGCSE'04*, pp.210–213 (2004).

[19] Parsons, D. and Haden, P.: Parsons Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses, *Proc. 8th Australasian Conference on Computing Education* (*ACE '06*), Vol.52, pp.157–163 (2006).

[20] Denny, P., Luxton-Reilly, A. and Simon, B.: Evaluating a New Exam Question: Parsons Problems, *Proc. 4th Intl. Workshop on Computing Education Research 2008* (*ICER'08*), pp.113–124 (Sep. 2008).

[21] Ericson, B.J., Margulieux, L.E. and Rick, J.: Solving Parsons Problems Versus Fixing and Writing Code, *Proc. 17th Koli Calling International Conference on Computing Education Research*, pp.20–29 (Nov. 2017).

[22] Cheng, N. and Harrington, B.: The Code Mangler: Evaluating Coding Ability Without Writing Any Code, *Proc. SIGCSE'17*, pp.123–128 (Mar. 2017).

[23] Kuno, Y.: "Information Study" as A Subject in University Entrance Examination, *Magazine of Information Processing Society of Japan*, Vol.55, No.4, pp.352–355 (Mar. 2014) (in Japanese).

Appendix

A.1 Problems for College Class Experiments

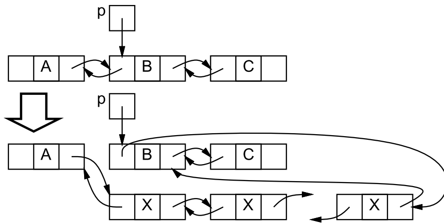
Figures A·1, **A·2**, **A·3**, and **A·4** show the problems C, D, E and F correspondingly.

© 2020 Information Processing Society of Japan

You are expected to create “List” Abstract Data Type by using doubly-linked list implementation. Write a function to insert `n` nodes with given `elementtype` value `x` at `position p`. Related data definitions and the function header are as follows.

```
struct node {
  elementtype element;
  struct node *next;
  struct node *previous;
}
typedef struct node *link;
typedef link list;
typedef link position;
void insertn(list L, position p,
    elementtype t, int n){
      (answer here)
}
```

For example, the following figure shows the change when `n` elementtype “X” nodes are inserted at `position p`.



(SP only) Choose code lines from the following choices and fill in corresponding symbols in the answer field. You can use the same line multiple times.

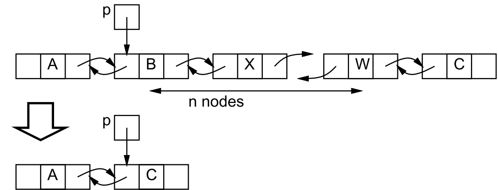
```
A struct node *a, *q;
B while(n>0){
C }
D a = (struct node*)
    malloc(sizeof(struct node));
E a->element = x;
F a->next = p;
G a->previous - q;
H p = a;
I p->previous = a;
J q = a;
K q = p->previous;
L q->next = a;
M n--;
```

Fig. A·1 Problem C.

You are expected to create “List” Abstract Data Type by using doubly-linked list implementation. Write a function to remove `n` nodes at `position p`. You can assume that more than `n` nodes exist at the specified position. Related data definitions and the function header are as follows.

```
struct node {
  elementtype element;
  struct node *next;
  struct node *previous;
}
typedef struct node *link;
typedef link list;
typedef link position;
void deleten(list L, position p,
    int n){
      (answer here)
}
```

For example, the following figure shows the change when `n` nodes are removed at `position p`.



(SP only) Choose code lines from the following choices and fill in corresponding symbols in the answer field. You can use the same line multiple times.

```
A link q;
B while(n>0){
C }
D n--;
E q=p->previous;
F q->next = p->next;
G q->next = p;
H p = p->next;
I p->previous->next = p->next;
J p->next->previous = p->previous;
K p->previous = q;
L p->next->previous = q;
```

Fig. A·2 Problem D.

© 2020 Information Processing Society of Japan

Write a function that accepts `recordtype` array `a`, `int n` and sort elements at index 1 through `n` of the array using selection sort. The following are the data definitions, a support function, and the function header.

```
typedef struct {
  keytype key;
  ...
} recordtype;
recordtype a[LIMIT]; // max n + 1
void swap(recordtype *x, recordtype *y){
  recordtype temp;
  temp = *x; *x = *y; *y = temp;
}
void selection_sort(recordtype
    a[], int n){
      (answer here)
}
```

(SP only) Choose code lines from the following choices and fill in corresponding symbols in the answer field. You can use the same line multiple times.

```
A int i, j, p;
B for(i=1;i<n;i++){
C for(i=1;i<=n;i++){
D for(j=i+1;j<=n;j++){
E for(j=i;j<=n;j++){
F for(j=i;j<n;j++){
G p = i;
H p = j;
K if(a[j].key<a[p].key){
L if(a[i].key<a[p].key){
M if(a[i].key<a[j].key){
O swap(&a[p],&a[i]);
P swap(&a[p],&a[j]);
Q swap(&a[i],&a[j]);
R }
```

Fig. A·3 Problem E.

Write a function that accepts `recordtype` array `a`, `int n` and sort elements at index 1 through `n` of the array using bubble sort. The following are the data definitions, a support function, and the function header.

```
typedef struct {
  keytype key;
  ...
} recordtype;
recordtype a[LIMIT]; // max n + 1
void swap(recordtype *x, recordtype *y){
  recordtype temp;
  temp = *x; *x = *y; *y = temp;
}
void bubble_sort(recordtype
    a[], int n){
  int i,j;
      (answer here)
}
```

(SP only) Choose code lines from the following choices and fill in corresponding symbols in the answer field. You can use the same line multiple times.

```
A int i, j, p;
B for(i=1;i<n;i++){
C for(i=1;i<=n;i++){
D for(j=i+1;j<=n;j++){
E for(j=i;j<=n;j++){
F for(j=i;j<n;j++){
G p = i;
H p = j;
K if(a[j].key<a[p].key){
L if(a[i].key<a[p].key){
M if(a[i].key<a[j].key){
O swap(&a[p],&a[i]);
P swap(&a[p],&a[j]);
Q swap(&a[i],&a[j]);
R }
```

Fig. A·4 Problem F.

Yasuichi Nakayama is a professor in the Graduate School of Informatics and Engineering at The University of Electro-Communications. He received his B.E., M.E., and D.Eng. degrees from The University of Tokyo in 1988, 1990, and 1993, respectively. His research interests include operating systems, parallel and distributed computing, and ICT education. He is a board member of the Information Processing Society of Japan.

Yasushi Kuno is a professor in the Graduate Scool of Informatics and Enginnering at The University of Electro-Comunications. He recieved his B.S., M.S., and D. Sci. degrees fron Tokyo Institute of Technology in 1979, 1981, and 1986 respectively. His research intersts include programming languages and ICT education.

Hiroyasu Kakuda received his B.S., M.S., D.Sci. degrees from Tokyo Institute of Technology in 1974, 1976, and 1982, respectively. He was an associate professor in the Department of Communication Engineering and Informatics at The University of Electro-Communications from 1992 to 2016. He is a part-time lecturer at Tokyo Metropolitan College of Industrial Technology. His research interests include human computer interaction, computers and education, Japanese document processing, and string manipulation. He is a member of IPSJ and ACM.

© 2020 Information Processing Society of Japan