

COS 126

Princeton University

Fall 2026

Midterm

Questions/points. This exam has 10 questions (Q0–Q9). You have 80 minutes to complete it. Q0 is the usual initialization question, a reminder to fill out this cover page (2 points). The main body of the exam (Q1–Q8) is worth 78 points, for a total of 80. Q9 is optional extra credit, worth up to 3 points; we recommend attempting it only if you have completed and checked over the rest of the exam.

Instructions. This exam is preprocessed by computer. Write neatly and legibly, using a dark pen or pencil. Put all answers (and nothing else) inside the designated spaces. *Fill in* bubbles completely (●). To change an answer, erase it completely and rewrite it. You may use the margins and the last page for scratch work, provided it does not interfere with boxes and bubbles for answers. Do not remove or unstaple any pages.

Resources. The exam is closed book, except that you may use a one-page reference sheet (8.5-by-11 paper, one side, handwritten by you). No electronic devices are permitted.

Honor Code. This exam is governed by Princeton’s Honor Code. Discussing the contents of this exam before solutions are posted violates the Honor Code.

Please complete the following information:

Name:

Margaret Hamilton

NetID:

mhamilton

Exam room:



McCosh 50



JRR-A17



Other:

Precept:

P01



P01A



P02



P03



P03A



P04



P04A



P05



P05A



P06



P10



P11



P13



Neighbor:

If the exam has not started yet, write the name of the person sitting on your right as you face the front of the room. If nobody is nearby on that side or if the exam has already started, write “location:” and your approximate seat location.

Tony Hoare

“I pledge my honor that I will not violate the Honor Code during this examination.”

I pledge my honor that I will not violate
the Honor Code during this examination.

Margaret Hamilton

Signature

(2 pts) **0. Initialization (2 points).**

In the designated spaces on the front of this exam:

- Write your name.
- Write your Princeton NetID (6–8 alphanumeric characters).
- Fill in the bubble corresponding to the room in which you are taking this exam, or write the room in the Other box.
- Fill in the bubble corresponding to your precept.
- Fill in the box labeled “Neighbor”.
- Write and sign the Honor Code pledge.

1. Java Language Properties (5 points).

Identify each statement as true or false by filling in the appropriate bubble.

true false

- (1/2 pt) (a) ☒ ☐ Comments in a Java program are ignored by the compiler.
The compiler discards comments; they are for human readers.
- (1/2 pt) (b) ☐ ☒ In Java's `main(String[] args)`, `args.length` is always at least 1.
With no command-line arguments, `args` is an empty array: `args.length` is 0.
- (1/2 pt) (c) ☐ ☒ A variable declared inside a `for` loop is accessible after the loop ends.
Its scope is the loop: a variable is accessible only within the block that declares it.
- (1/2 pt) (d) ☒ ☐ Two functions in the same `.java` file can have the same name.
Overloading: allowed as long as their parameter lists differ.
- (1/2 pt) (e) ☒ ☐ A `void` function may contain one or more `return` statements.
`return;` (with no value) ends a `void` function early, and may appear several times.
- (1/2 pt) (f) ☒ ☐ A `private` function can be called by any other function in the same `.java` file.
`private` limits access to code in the same class, i.e., the same `.java` file.
- (1/2 pt) (g) ☐ ☒ An array created with `new int[8]` has its last element at index 8.
Its indices run from 0 to 7, so its last element is at index 7.
- (1/2 pt) (h) ☒ ☐ A function that returns a value may also produce side effects.
For example, it may print, or modify an array argument, and also return a value.
- (1/2 pt) (i) ☐ ☒ Two functions with the same name must have different return types.
They must have different parameter lists; their return types may be the same.
- (1/2 pt) (j) ☐ ☒ You must declare each variable to have a specific type, but you can change its type with a subsequent declaration statement.
A variable's type is fixed; declaring it again in the same scope is a compile-time error.

2. Java Expressions (10 points).

Assume the following variables have been declared and initialized:

```
int i = 4;
int j = 6;
double d = 2.0;
String s = "126";
```

For each expression, write the letter of the best-matching value in the Answer column. “error” means a compile-time error or runtime exception. You may use each letter once, more than once, or not at all.

Answer	Expression	Options
		A. false
(2 pts)	(a) G <code>i + j / i</code> Integer division first: <code>6 / 4</code> is 1, so <code>4 + 1 = 5</code> .	B. true
(2 pts)	(b) F <code>(double) i / (i - d)</code> The cast applies to <code>i</code> : <code>4.0 / (4 - 2.0)</code> is 2.0.	C. 0
(1 pt)	(c) E <code>i % j - j % i</code> <code>4 % 6</code> is 4 and <code>6 % 4</code> is 2, so <code>4 - 2 = 2</code> (an int).	D. 0.0
(1 pt)	(d) H <code>Integer.parseInt(s) / i</code> <code>126 / 4</code> is 31: integer division.	E. 2
(1 pt)	(e) I <code>s + i + j</code> Left to right: <code>"126" + 4</code> is <code>"1264"</code> , then <code>+ 6</code> gives <code>"12646"</code> : string concatenation, not addition.	F. 2.0
(1 pt)	(f) D <code>j + 0.0 - j</code> <code>6 + 0.0</code> is 6.0 (a double), and <code>6.0 - 6</code> is 0.0.	G. 5
(1 pt)	(g) A <code>i < j && i < d</code> <code>4 < 6</code> is true but <code>4 < 2.0</code> is false, so <code>&&</code> gives false.	H. 31
(1 pt)	(h) K <code>s / i</code> <code>/</code> is not defined for a String and an int: a compile-time error.	I. "12646"
		J. "12610"
		K. error

3. Loops, Tracing, and Performance (11 points).

In the box next to each code fragment below, write the output of that code. The fragments differ only in the lines inside the dashed boxes.

(3 pts) (a)

10

Fragment 1

```
int n = 4;
int count = 0;

for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++)
        count++;

StdOut.println(count);
```

For $i = 1, 2, 3, 4$, the inner loop runs j from i to 4: $4 + 3 + 2 + 1 = 10$ times.

(3 pts) (b)

2

Fragment 2

```
int n = 4;
int count = 0;

int i = n;
while (i > 1) {
    count++;
    i = i / 2;
}

StdOut.println(count);
```

i goes 4, 2, 1: the body runs for $i = 4$ and $i = 2$, then the loop stops, since 1 is not greater than 1. So count is 2.

(2 pts) (c)

6

Fragment 3

```
int n = 4;
int count = 0;

for (int i = 0; i < n; i++)
    count += i;

StdOut.println(count);
```

count adds $i = 0, 1, 2, 3$:
 $0 + 1 + 2 + 3 = 6$.

Suppose that, in each fragment above, n were instead initialized from a command-line argument:

```
int n = Integer.parseInt(args[0]);
```

What is the order of growth of the running time of each fragment as a function of n ? Fill in one bubble for each fragment.

		constant $\Theta(1)$	logarithmic $\Theta(\log n)$	linear $\Theta(n)$	linearithmic $\Theta(n \log n)$	quadratic $\Theta(n^2)$	cubic $\Theta(n^3)$	exponential $\Theta(c^n)$
(1 pt) (d)	Fragment 1	☐	☐	☐	☐	☒	☐	☐
(1 pt) (e)	Fragment 2	☐	☒	☐	☐	☐	☐	☐
(1 pt) (f)	Fragment 3	☐	☐	☒	☐	☐	☐	☐

(d) The inner loop runs $n + (n - 1) + \dots + 1 = n(n + 1)/2$ times in all: quadratic.

(e) i is halved each time, so the loop runs about $\log_2 n$ times: logarithmic.

(f) One loop of n iterations, each doing a constant amount of work: linear.

4. RandomWalkers (7 points).

Recall `RandomWalkers` from Assignment 1. The program simulates `trials` random walks of `n` steps each, starting from the origin $(0, 0)$, as illustrated below. Each step moves one unit north, south, east, or west at random. After each trial, the squared distance from the origin is recorded and the walker returns to the origin. The program prints the mean squared distance over all trials.

Fill in the bubble for true or false.

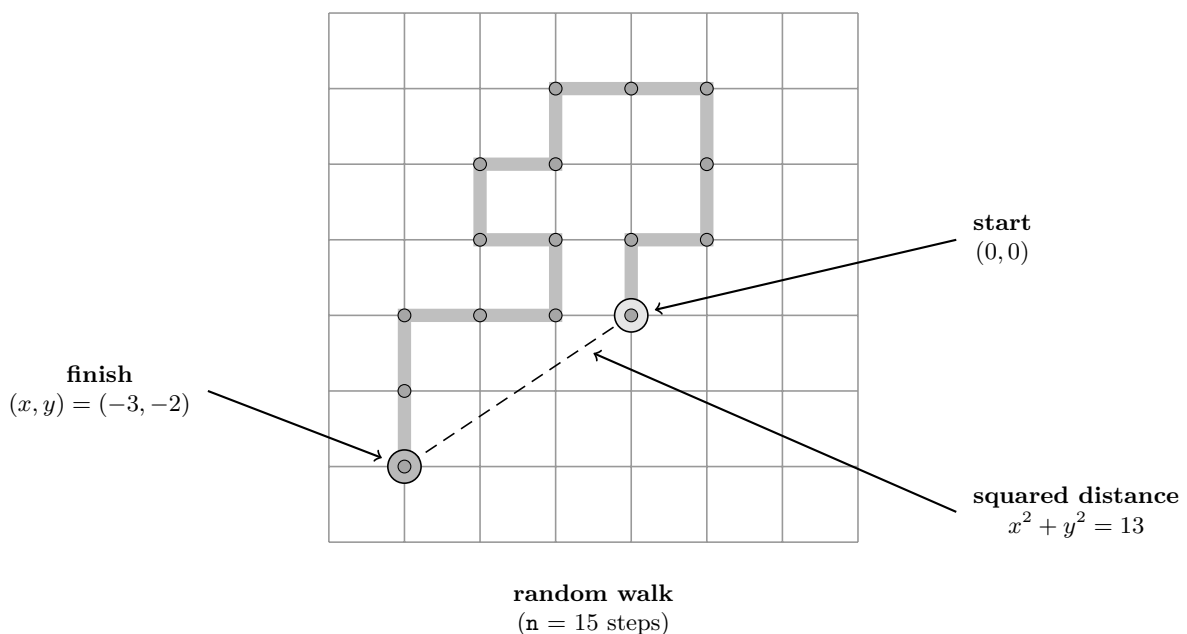
true false

- (1 pt) (a) ☐ ☒ During a single trial, an array is required to find the farthest distance from the origin reached at any step.
Keep the largest $x^2 + y^2$ seen so far in one variable, updated after each step; no array is needed.
- (1 pt) (b) ☒ ☐ If `n` is odd, a trial can never end at the origin.
Each step changes $x + y$ by ± 1 , so after an odd number of steps $x + y$ is odd, never 0.
- (1 pt) (c) ☒ ☐ After `n` steps, the squared distance from the origin is at most n^2 .
 $|x| + |y| \leq n$, so $x^2 + y^2 \leq (|x| + |y|)^2 \leq n^2$, with equality when every step goes the same way.
- (1 pt) (d) ☒ ☐ If the walker is not reset to the origin between trials, the printed mean squared distance tends to be larger.
Each trial would continue the previous walk, so squared distances keep growing.
- (1 pt) (e) ☐ ☒ `(int) Math.random() * 4` produces each of 0, 1, 2, and 3 with probability 1/4.
The cast applies first, so it is always 0; use `(int) (Math.random() * 4)`.

- (2 pts) (f) What is the smallest possible squared distance from the origin after exactly 5 steps?

1

5 is odd, so the walk cannot end at the origin (see b above), but it can end one step away, e.g. at $(1, 0)$: $1^2 + 0^2 = 1$.



5. Standard Input, Redirection, and Piping (10 points).

The two programs below are correct. `Sum` reads integers from standard input and prints their sum. `Filter` takes two command-line arguments `lo` and `hi`, reads integers from standard input, and prints only those in the range `[lo, hi]` (inclusive), one per line. File `data.txt` contains just what is shown.

```
Sum.java
public class Sum {
    public static void main(String[] args) {
        int total = 0;
        while (!StdIn.isEmpty()) {
            total += StdIn.readInt();
        }
        StdOut.println(total);
    }
}
```

```
data.txt
3 8 1 5 9 4
```

```
Filter.java
public class Filter {
    public static void main(String[] args) {
        int lo = Integer.parseInt(args[0]);
        int hi = Integer.parseInt(args[1]);
        while (!StdIn.isEmpty()) {
            int x = StdIn.readInt();
            if (x >= lo && x <= hi)
                StdOut.println(x);
        }
    }
}
```

For each **command**, write the **output** in the box.

- Write “**error**” if the command produces an error.
- Write “**blank**” if there is no output.

Command	Output
(1 pt) (a) <code>java-introcs Sum < data.txt</code> Simply sums the integers in the data file: $3 + 8 + 1 + 5 + 9 + 4 = 30$.	30
(1 pt) (b) <code>java-introcs Filter 4 < data.txt</code> <code>Filter</code> requires 2 arguments and here only 1 is provided: <code>args[1]</code> is out of bounds.	error
(2 pts) (c) <code>java-introcs Filter 4 8 < data.txt java-introcs Sum</code> <code>Filter</code> prints 8, 5, 4 (one per line), which <code>Sum</code> adds: $8 + 5 + 4 = 17$.	17
(2 pts) (d) <code>java-introcs Filter 6 7 < data.txt java-introcs Sum</code> No integer in the file is in <code>[6, 7]</code> , so <code>Filter</code> prints nothing, and <code>Sum</code> prints 0.	0
(2 pts) (e) <code>java-introcs Sum < data.txt java-introcs Filter 1 9</code> <code>Sum</code> prints 30, which is outside <code>[1, 9]</code> , so <code>Filter</code> prints nothing.	blank
(2 pts) (f) <code>java-introcs Filter 1 5 < data.txt java-introcs Filter 3 9 java-introcs Sum</code> <code>Filter 1 5</code> prints 3, 1, 5, 4; <code>Filter 3 9</code> keeps 3, 5, 4; <code>Sum</code> prints $3 + 5 + 4 = 12$.	12

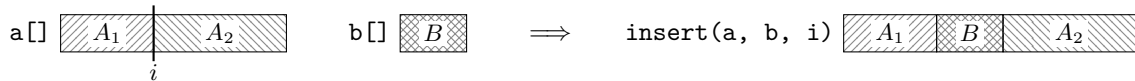
6. AudioCollage API (13 points).

Recall from `AudioCollage.java` (Assignment 3) that a sound is represented as an array of `double` values and that the library includes the function `merge()`. Suppose that the `AudioCollage` library also includes a new function `cut()` that cuts out a portion of an array, starting at `start` and containing `k` elements:

```
// Returns a new array that is the concatenation of a[] followed by b[].
public static double[] merge(double[] a, double[] b)

// Returns a new array containing the k elements of a[] starting at a[start].
// For example, cut(a, 2, 3) returns { a[2], a[3], a[4] }.
// Assumes start >= 0, k >= 0, and start + k <= a.length.
public static double[] cut(double[] a, int start, int k)
```

Using `merge()` and `cut()`, implement a new function `insert()` that inserts one sound into the middle of another. Suppose `a[]` consists of two parts: A_1 followed by A_2 (which starts at index `i`), and suppose `b[]` is B . Then `insert(a, b, i)` returns A_1 , followed by B , followed by A_2 , like this:



- (6 pts) (a) Fill in the body of `insert()`. Use `merge()` and `cut()`, but **DO NOT** use loops or `StdAudio`.

```
// Returns a new array consisting of a[0], ..., a[i-1], followed by
// b[], followed by a[i], ..., a[a.length-1].
// Assumes a[] and b[] are not null, and 0 <= i <= a.length.
public static double[] insert(double[] a, double[] b, int i) {
```

```
    double[] a1 = cut(a, 0, i);
    double[] a2 = cut(a, i, (a.length - i));
    return merge(merge(a1, b), a2);

    // Grading note: also accept one-line versions, such as
    // return merge(merge(cut(a, 0, i), b), cut(a, i, a.length - i));
    // and other valid nestings, such as merge(a1, merge(b, a2)).
```

```
}
```

- (3 pts) (b) What is the length of the array returned by `insert(a, b, i)`? Give your answer in terms of `a.length` and `b.length`.

`a.length + b.length`
 A_1 and A_2 together are all of `a[]`, and B is all of `b[]`.

- (3 pts) (c) Write an expression that calls `insert()` and returns the same result as `merge(a, b)`.

`insert(a, b, a.length) or insert(b, a, 0)`
 Insert `b[]` at the end of `a[]` (so A_2 is empty), or insert `a[]` at the start of `b[]` (so A_1 is empty).

- (1 pt) (d) What is the order of growth of the running time of `insert()` as a function of n , where n is the length of the array it returns? Assume `merge()` and `cut()` take time proportional to the length of the array they return. Fill in the bubble.

constant $\Theta(1)$	logarithmic $\Theta(\log n)$	linear $\Theta(n)$	linearithmic $\Theta(n \log n)$	quadratic $\Theta(n^2)$	cubic $\Theta(n^3)$	exponential $\Theta(c^n)$
☐	☐	☒	☐	☐	☐	☐

`insert()` makes a constant number of calls (two to `cut()` and two to `merge()`), and each returns an array of length at most n : linear.

7. Delannoy Recursion and Memoization (9 points).

Recall the Delannoy numbers from Assignment 4. $D(m, n)$ is the number of paths from $(0, 0)$ to (m, n) using steps north, east, and northeast:

$$D(m, n) = \begin{cases} 1 & \text{if } m = 0 \\ 1 & \text{if } n = 0 \\ D(m-1, n) + D(m, n-1) + D(m-1, n-1) & \text{otherwise} \end{cases}$$

A *brute-force* implementation calls itself directly using this recurrence, like `DelannoyBrute`. A *memoized* implementation saves each computed value in a table and reuses it, like `DelannoyMemo`.

(2 pts) (a) What is the value of $D(2, 1)$?

5

$D(2, 1) = D(1, 1) + D(2, 0) + D(1, 0) = 3 + 1 + 1 = 5$,
where $D(1, 1) = D(0, 1) + D(1, 0) + D(0, 0) = 3$.

(3 pts) (b) When the brute-force implementation computes $D(2, 2)$, how many times does it compute $D(1, 1)$?

3

$D(2, 2)$ calls $D(1, 1)$ directly, and so do its calls
 $D(1, 2)$ and $D(2, 1)$: 3 times in all.

Fill in the bubble for true or false.

true false

(1 pt) (c) ☒ ☐

$D(m, n) = D(n, m)$ for all $m \geq 0$ and $n \geq 0$.

The recurrence and base cases are symmetric in m and n : reflect each path across the diagonal.

(1 pt) (d) ☐ ☒

Swapping the order of the three recursive calls changes the value returned.

Addition is commutative: the sum is the same in any order.

(1 pt) (e) ☐ ☒

Memoization changes the value returned, not just the running time.

Memoization only reuses values already computed: the same values brute force computes.

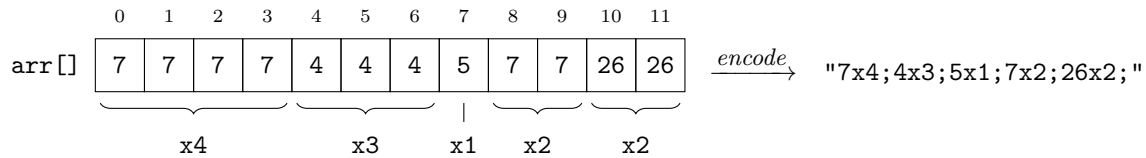
(1 pt) (f) ☒ ☐

With memoization, the recursive case for each pair of arguments (i, j) executes at most once.

The first call saves the value in the table; later calls for (i, j) return it with no further recursive calls.

8. Run-Length Encoding (13 points).

Run-length encoding compresses data by replacing each run of repeated values with the value and the length of the run. For example, this integer array will be encoded in a `String` as follows:



Each run is represented as the value, followed by the letter `x`, followed by the run length, then a semicolon.

We will need a helper function `runLength(arr, i)` that returns the length of the run that starts at `arr[i]`. It ignores any entries before `arr[i]`. Examples for the array above are in the box here. ↩

- (1 pt) (a) For the array above, what is `runLength(arr, 4)`?

3

`arr[4]` starts the run 4, 4, 4 (indices 4–6), so its length is 3.

Examples

`runLength(arr, 0) → 4`
`runLength(arr, 2) → 2`
`runLength(arr, 3) → 1`

- (1 pt) (b) For the array above, what is `runLength(arr, 6)`?

1

`arr[6]` is the last 4 of that run, and entries before `arr[i]` are ignored, so the length is 1.

- (7 pts) (c) Write the body of `runLength()` using a loop.

// Assumes 0 ≤ i < arr.length.

`private static int runLength(int[] arr, int i) {`

```
// Count forward while the next entry matches arr[i].
int count = 1;
while (i + count < arr.length && arr[i + count] == arr[i])
    count++;
return count;

-- or --

// Advance a second index j past the end of the run.
int j = i + 1;
while (j < arr.length && arr[j] == arr[i])
    j++;
return j - i;
```

`}`

(d)–(e) Suppose `arr[]` holds the results of rolling a six-sided die 1,000 times: the array has length 1,000, and each element is an integer from 1 to 6. What are the **smallest** and **largest** possible numbers of characters in the encoded `String`?

- (2 pts each) (d) *smallest:*

7

- (e) *largest:*

4000

(d) All rolls equal: one run, e.g. "6x1000;"

(e) No equal neighbors: 1,000 runs.

Like "3x1;2x1;4x1;... (996 more) ...2x1;"

Optional Extra Credit

Everything on this page is optional. Work on it only after finishing the rest of the exam.

9. Run-Length Encoding, Continued (up to 3 bonus points).

This question continues Question 8.

(a) (1 point) Write `runLength()` again, this time recursively, with no loops.

```
// Assumes 0 <= i < arr.length.
private static int runLength(int[] arr, int i) {

    // Base: if arr[i] ends the run, its length is 1.
    // Otherwise, add 1 to the length of the run starting at i+1.
    if (i + 1 == arr.length || arr[i + 1] != arr[i]) return 1;
    return 1 + runLength(arr, i + 1);

    -- or --

    // Check the two conditions separately (the end first).
    if (i + 1 == arr.length) return 1;    // reached end
    if (arr[i + 1] != arr[i]) return 1;    // entries differ
    return 1 + runLength(arr, i + 1);    // same run continues at i+1

}
```

(b) (2 points) Using `runLength()` as a helper, complete the body of `encode()` so that `result` holds the run-length encoding of `arr[]`, such as `"7x4;4x3;5x1;7x2;26x2;"`.

```
// Assumes arr.length >= 1.
public static String encode(int[] arr) {
    String result = "";    // initialize to the empty string

    int i = 0;
    while (i < arr.length) {
        int run = runLength(arr, i);
        result += arr[i] + "x" + run + ";";
        i += run;
    }

    return result;    // return the encoding
}
```

This page is intentionally left blank. You may use it for scratch work.