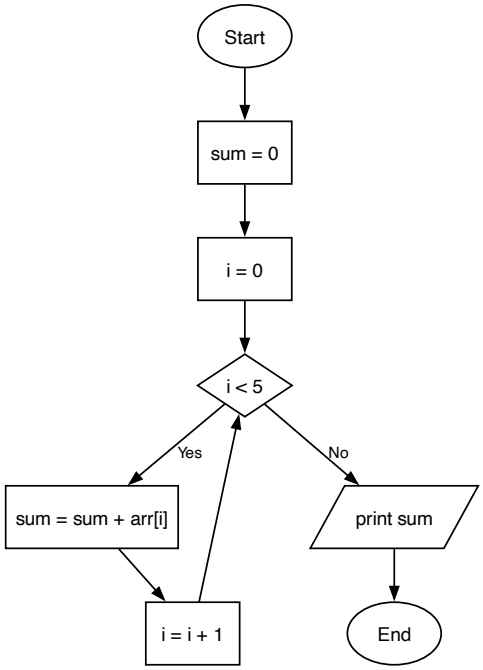
Problem 1.

> *This program adds up all the numbers in a list. It keeps a running total in sum and a position marker i, and walks through the list one item at a time — from position 0 to the last — adding each item to the total as it goes.*

Trace this algorithm, taking the list *arr* to be [4, 7, 2, 9, 5], and complete the trace table.

Reminder: the items of a list have positions counted from 0, so the list [4, 7, 2, 9, 5] has 5 items in positions 0 to 4 — position 0 holds 4, position 1 holds 7, and so on. arr[i] means the item at position i, so arr[0] is the first item.



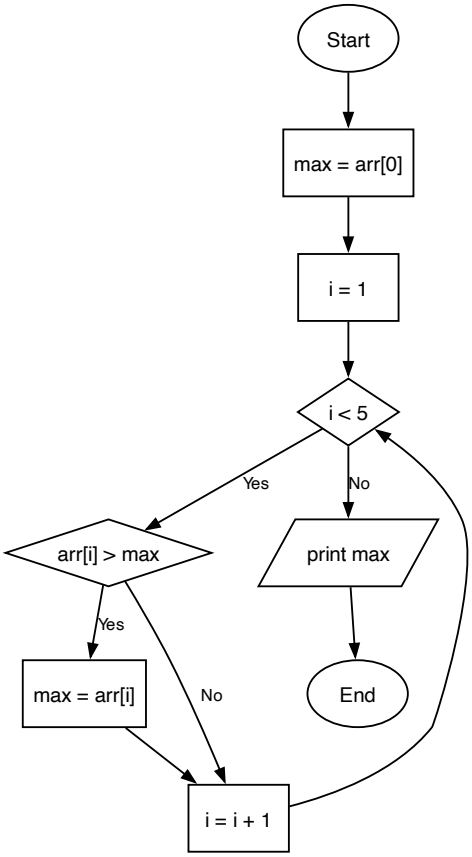
sum	i	output

Problem 2.

This program finds the largest number in a list. It begins by assuming the first item (position 0) is the biggest, then walks through the rest of the list: whenever it meets an item bigger than the best seen so far, it remembers that one instead. max holds the largest value found so far.

Trace this algorithm, taking the list *arr* to be [3, 9, 2, 12, 7], and complete the trace table.

Reminder: the items of a list have positions counted from 0, so the list [4, 7, 2, 9, 5] has 5 items in positions 0 to 4 — position 0 holds 4, position 1 holds 7, and so on. arr[i] means the item at position i, so arr[0] is the first item.



max	i	output

Problem 3.

> *Go through a list of numbers from start to finish and, for each one, print that number multiplied by 2 — for example, for the list [2, 5, 0, 1, 4] the program prints 4, 10, 0, 2, 8.*

(i) Before drawing anything, work out by hand what the program should print for each input below. (Once you have drawn your flowchart, trace it on these same inputs and check that it agrees with your answers here.)

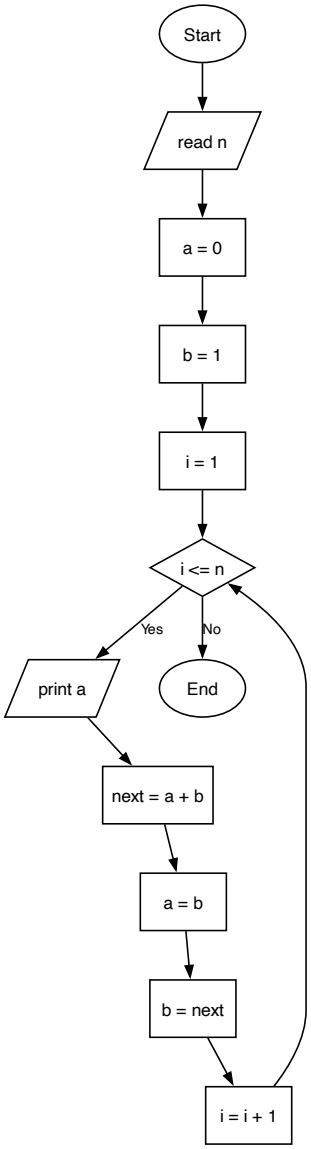
input *arr*	output
(3, 8, 1, 6, 4)	
(10, 0, 5, 2, 7)	

(ii) Now draw a flowchart for the algorithm.

Problem 4.

The Fibonacci sequence starts 0, 1 and then every number after that is the sum of the two before it (0+1=1, 1+1=2, 1+2=3, and so on). This program prints the first n numbers of the sequence. It keeps the last two numbers in a and b: each time round the loop it prints a, works out the next number (a + b) before changing anything, then slides the pair along so a and b become the next two numbers.

Trace this algorithm for $n = 7$, and complete the trace table.



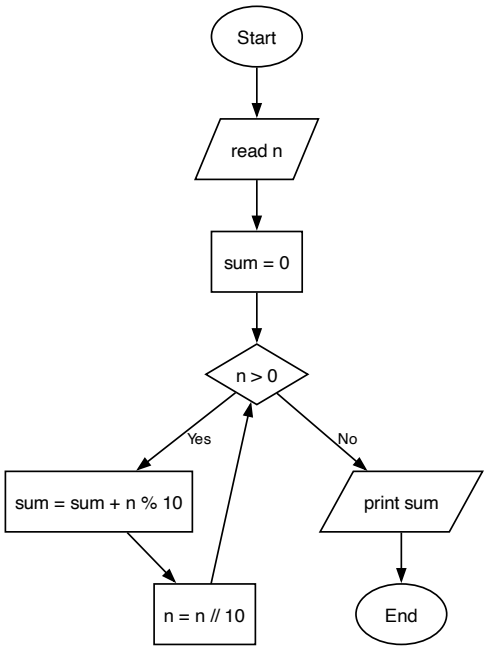
n	a	b	i	next	output

Problem 5.

This program reads a whole number and adds up its digits. It peels off the last digit each time (using the remainder after dividing by 10), adds it to a running total, then drops that digit and repeats until nothing is left.

Trace this algorithm for $n = 4072$, and complete the trace table.

Reminder: n % 10 is the remainder when n is divided by 10 (the last digit of n), and n // 10 is n divided by 10 with the remainder thrown away (which drops the last digit) — so 4072 % 10 is 2 and 4072 // 10 is 407.



n	sum	output

Problem 6.

> *Ask the user for a whole number n and print how many digits it has — for example 4072 has 4 digits and 9 has 1 digit. (Hint: you can drop the last digit of a number by doing n // 10, i.e. dividing by 10 and throwing away the remainder. Keep doing that, counting as you go, until the number reaches 0.)*

(i) Before drawing anything, work out by hand what the program should print for each input below. (Once you have drawn your flowchart, trace it on these same inputs and check that it agrees with your answers here.)

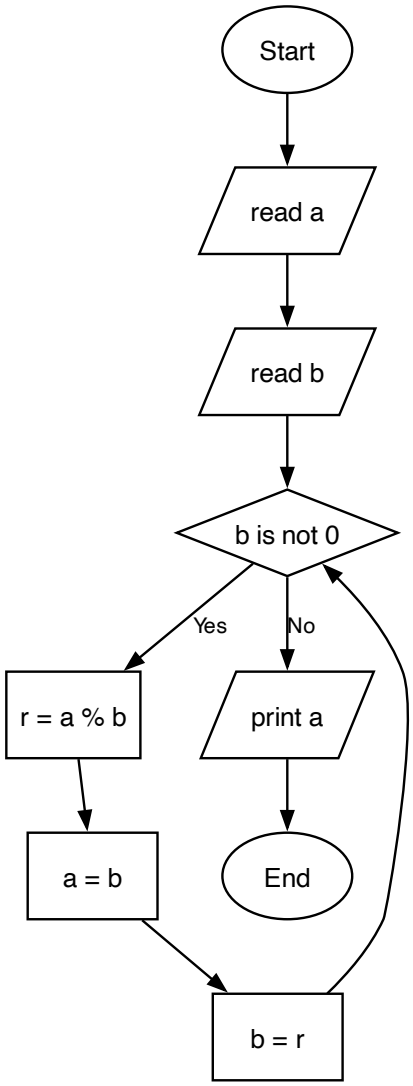
input n	output
4072	
538	
9	

(ii) Now draw a flowchart for the algorithm.

Problem 7.

This is Euclid's method for the greatest common divisor (the largest number that divides both a and b exactly). Each step replaces the pair (a, b) with (b, remainder of a divided by b); when the remainder reaches 0, the answer is the value left in a.

Trace this algorithm for $a = 48, b = 36$, and complete the trace table.
Reminder: a % b is the remainder when a is divided by b (for example 48 % 36 is 12, and 36 % 12 is 0).



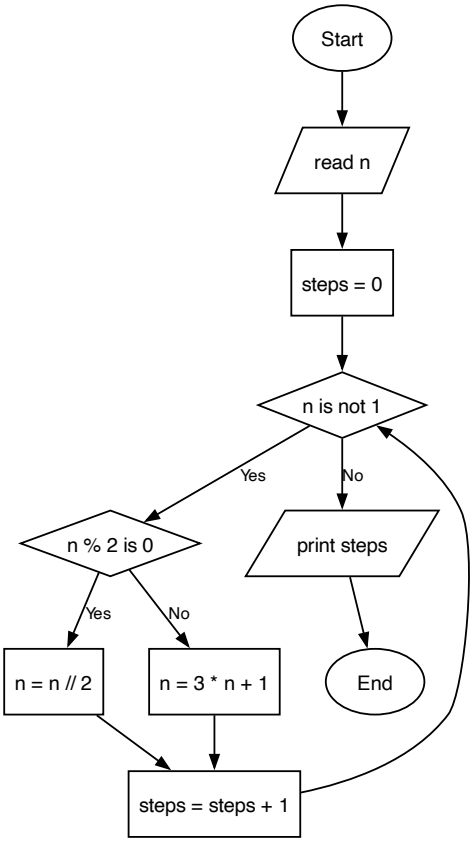
a	b	r	output

Problem 8.

Start from a whole number n and repeat one rule until you reach 1: if n is even, halve it; if n is odd, replace it with 3 × n + 1. This program counts how many steps that takes. (No one has ever found a starting number that fails to reach 1 — but no one has proved it always does, either!)

Trace this algorithm for $n = 6$, and complete the trace table.

Reminder: n % 2 is the remainder when n is divided by 2, so n % 2 is 0 exactly when n is even; n // 2 is n halved with any remainder dropped.



n	steps	output

Problem 9.

> *Ask the user for a whole number n and print the number you get by reversing its digits — so 825 becomes 528 and 1900 becomes 91 (leading zeros just disappear). (Hint: peel off the last digit of n with n % 10, and grow a result by doing rev = rev × 10 + that digit each time, while dropping the digit from n with n // 10.)*

(i) Before drawing anything, work out by hand what the program should print for each input below. (Once you have drawn your flowchart, trace it on these same inputs and check that it agrees with your answers here.)

input n	output
825	
1900	
7	

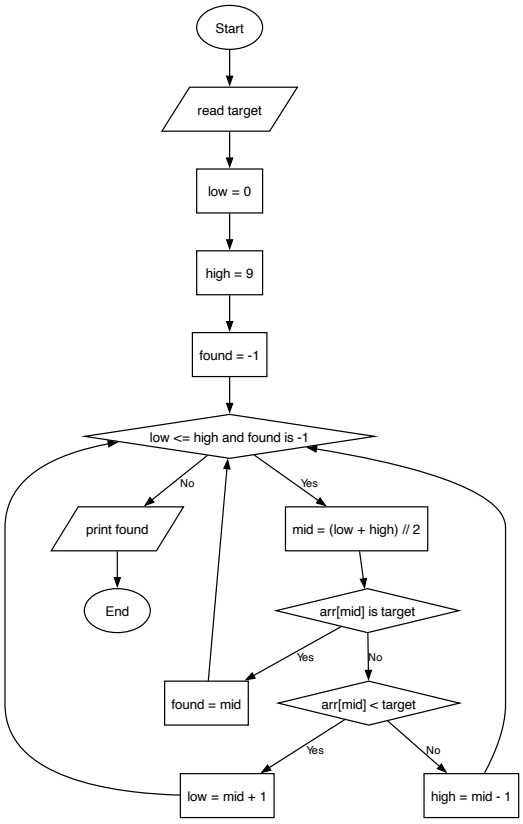
(ii) Now draw a flowchart for the algorithm.

Problem 10.

Binary search finds where a target value sits in a sorted list without checking every item. It tracks a window with two markers, low and high (the positions still worth searching, numbered from 0). Each step looks at the middle position mid: if that item is the target we are done; if it is too small the target must be to the right, so move low up; if it is too big, move high down. found holds the position of the target, or stays -1 until it is located.

Trace this algorithm for *target* = 23, taking the list *arr* to be [2, 5, 8, 12, 16, 23, 38, 56, 72, 91], and complete the trace table.

Reminder: arr[mid] means the item at position mid of the list (positions start at 0, so arr[0] is 2); (low + high) // 2 is the middle position, dividing by 2 and dropping any remainder; and the loop keeps going while low <= high and found is still -1.



target	low	high	found	mid	output