CS1113
Simple Algorithms

Lecturer:

Professor Barry O’Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

http://osullivan.ucc.ie/teaching/cs1113/

Simple Algorithms

a review of algorithms
writing down algorithms
informal analysis

linear search
binary search

Algorithms: a reminder

From lecture 02:

The Fundamental Concept of Computing

- Anything a computer does is based on carrying out a sequence of steps.

An algorithm is an ordered, deterministic, executable, terminating set of instructions

- "ordered" does not mean just a simple sequence
  - we can have branches, loops and function calls, as long as we know which instruction to carry out at each step
- we must carry out the instructions in the specified order
  - we can't jump ahead to work out what needs to be done
- the instructions must be clear, unambiguous, executable, and *complete*
  - they should cover all possible cases

Writing down algorithms

- For it to be useful, an algorithm must be written in some language that is understood by whoever or whatever is going to read it
  - You are learning how to express algorithms in Python, so that they can be executed by computers
- We are interested in something more fundamental: the essence of the algorithm, independent of the computer or architecture or programming language
  - but we still have to write it in some consistent style so that we can understand what the algorithm says and how efficient it would be if it were executed

Example: finding the minimum value in a sequence

Problem: given a finite sequence of integers, find the smallest integer in the sequence

Input: a sequence x[1], x[2], x[3], x[4], ..., x[n] of integers
Output: an integer min

```
1. min :=  x[1]
2. for each value of i from 2 to n
3.    if x[i] < min
4.    then min := x[i]
5. return min
```

Example: executing the algorithm

Suppose our input is the sequence <5,8,3,23,4,-2,1,0>

n: 8

x[1]=5
x[2]=8
x[3]=3
x[4]=23
x[5]=4
x[6]=-2
x[7]=1
x[8]=0

Input: a sequence x[1], x[2], x[3], x[4], ..., x[n] of integers
Output: an integer min

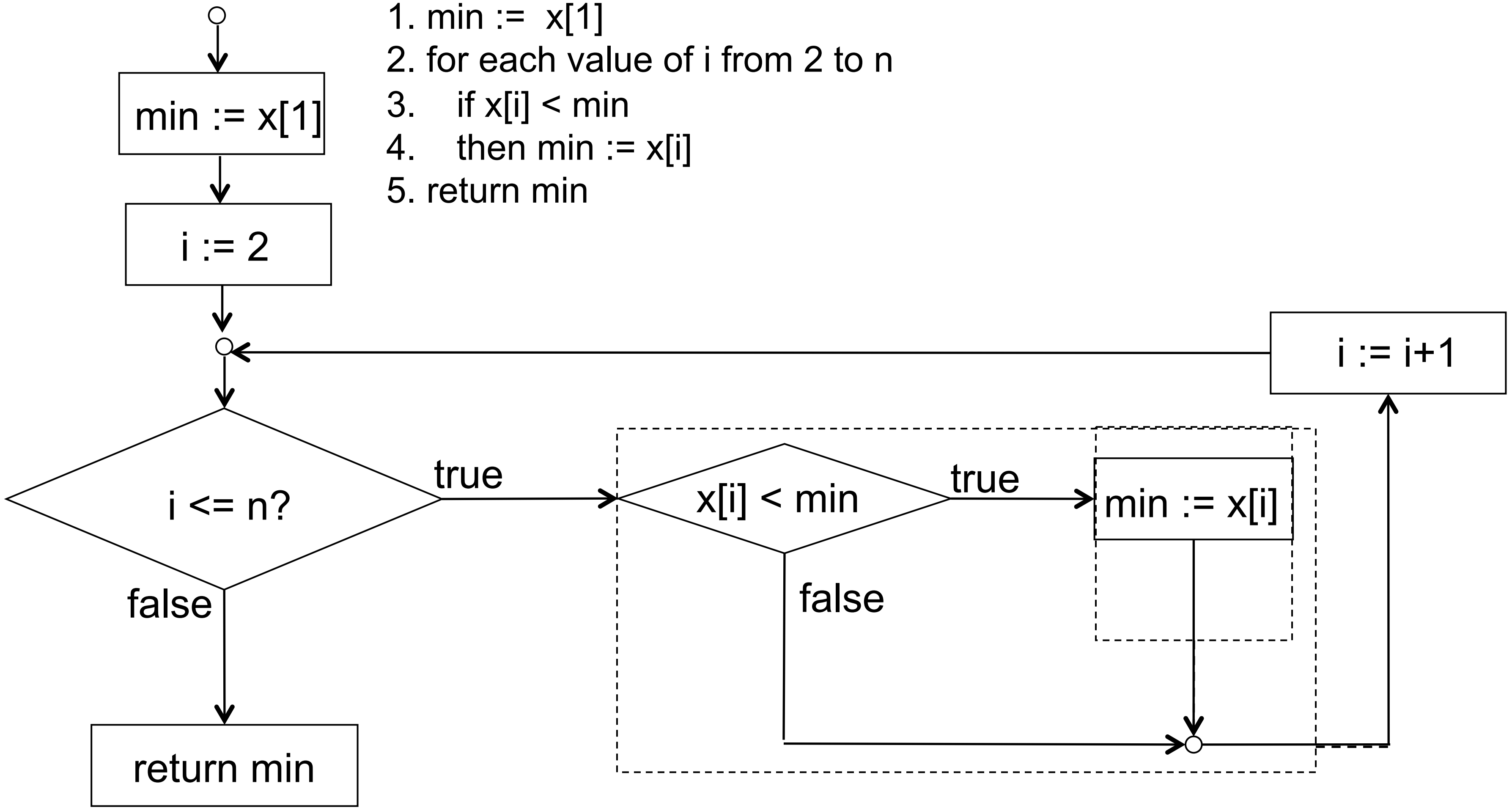
```
1. min :=  x[1]
2. for each value of i from 2 to n
3.     if x[i] < min
4.     then min := x[i]
5. return min
```

	i: 2	i: 3	i: 4	i: 5	i: 6	i: 7	i: 8
x[1]: 5	x[i]: 8	x[i]: 3	x[i]: 23	x[i]: 4	x[i]: -2	x[i]: 1	x[i]: 0
min: 5		min: 3			min: -2		

Visualising with a flowchart

Input: a sequence x[1], x[2], x[3], x[4], ..., x[n] of integers
Output: an integer min

```
1. min :=  x[1]
2. for each value of i from 2 to n
3.     if x[i] < min
4.     then min := x[i]
5. return min
```



Informal language for algorithms

Always state the input and output

use variable names

```
Input: a sequence x[1], x[2], x[3], x[4], ..., x[n] of integers
Output: an integer min

1. min :=  x[1]
2. for each value of i from 2 to n
3.    if x[i] < min
4.    then min := x[i]
5. return min
```

for loop – go round the loop this many times

indent three spaces inside a branch or loop

use ":=" for assignment – assign the value of x[1] to min

if-then-else: if test is true, do "then" part; if test is false, do "else" part if there is one; then move on to next line

Example: informal analysis of the algorithm

An algorithm is an ordered, deterministic, executable, terminating set of instructions

- **Correctness** Does it do what we want?
  - at first, min is set to the first element; each time we go round the loop, min is set to the smallest we have seen so far; so at the end, min tells us the smallest element.

this is called the "*loop invariant*" – a property that is always true at the end of each iteration of the loop

- **Termination** Does it terminate?
  - the input is finite, and the *for* loop says increment *i* each time we go round the loop until we reach the input length. Inside the loop we don't change *i*, and inside each loop we do a finite number of steps, so the *for* loop definitely terminates, and so the algorithm terminates
- **Complexity** How many steps?
  - 1 initial assignment step, and then once round the for loop for each other element of the sequence (so *n-1* times round the loop); inside the loop, we do 1 test, and we may do 1 assignment of a value to a variable. So at most, we need *n* assignments, and *n-1* tests.

Example: finding the average value in a sequence

Problem: given a finite sequence of integers, find the average value of the sequence

Input: sequence x[1], x[2], x[3], ..., x[n] of integers
Output: a decimal number ave

```
1. sum :=  0
2. for each value of i from 1 to n
3.    sum := sum+x[i]
4. ave := sum/n
5. return ave
```

Example: informal analysis of the algorithm

- **Correctness** Does it do what we want?
  - at first, sum is set to 0; each time we go round the loop, sum is set to the total of all values we have seen; so at the end of the loop, sum is the total of all values; we then divide by the number of values to get the average.
- **Termination** Does it terminate?
  - the input is finite, and the *for* loop says increment *i* each time we go round the loop until we reach the input length. Inside the loop we don't change *i*, and inside each loop we do a single addition and a single assignment, so the *for* loop terminates, and so the algorithm terminates
- **Complexity** How many steps?
  - 1 initial assignment step, and then once round the for loop for each element of the sequence (so *n* times round the loop); inside the loop, we do 1 addition, and we do 1 assignment of a value to a variable. After the loop, we do one division and 1 assignment. So we need *n*+2 assignments, and *n* additions, and 1 division.

Search

Often, we will need to find a particular value somewhere in a sequence of values.

- searching for a word in a spell-checker
- searching for a book in a library catalogue
- searching for a name in an address book
- searching for a keyword in a web page
- searching for a DVD on an e-commerce site

If we have to do this type of search often inside a program, then the speed at which we can do it is important.

We will look at some different ways to search, and see how long they take on average.

Linear Search

The simplest way to search a sequence is simply to consider each element in turn until we find the one we want.

Input: sequence *x[1], x[2], x[3], ..., x[n]* of objects
Input: *y*, the object we want to find
Output: *pos*, the position of *y* in the sequence, or 0 if it isn't there

```
1. pos := 0
2. i := 1
3. while (pos == 0 and i <= n)
4.     if (x[i] == y)
5.     then pos := i
6.     else i := i+1
7. return pos
```

"while" loop – similar to the "for" loop, but we now keep going round as long as the test is true

Informal analysis of linear search

- does the algorithm do what we want?
  - we consider each object in the sequence in turn, until we find the object we want, exit the loop and report the position, or we reach the end and report 0, so yes.
- does it terminate?
  - the while loop terminates if *pos* != 0 or *i* > *n*. *i* starts at the value 1. Inside the loop, each time we either increase the value of *i*, or we assign the value of *i* to *pos*. *i* is never 0, so at some stage, we will either assign a non-zero value to *pos*, or we will assign a value bigger than *n* to *i*. So the loop will stop, and so will the algorithm.
- how many steps does it take?
  - at worst, if the object is not in the sequence, we will examine every object, and so will take *n* steps.

Binary search

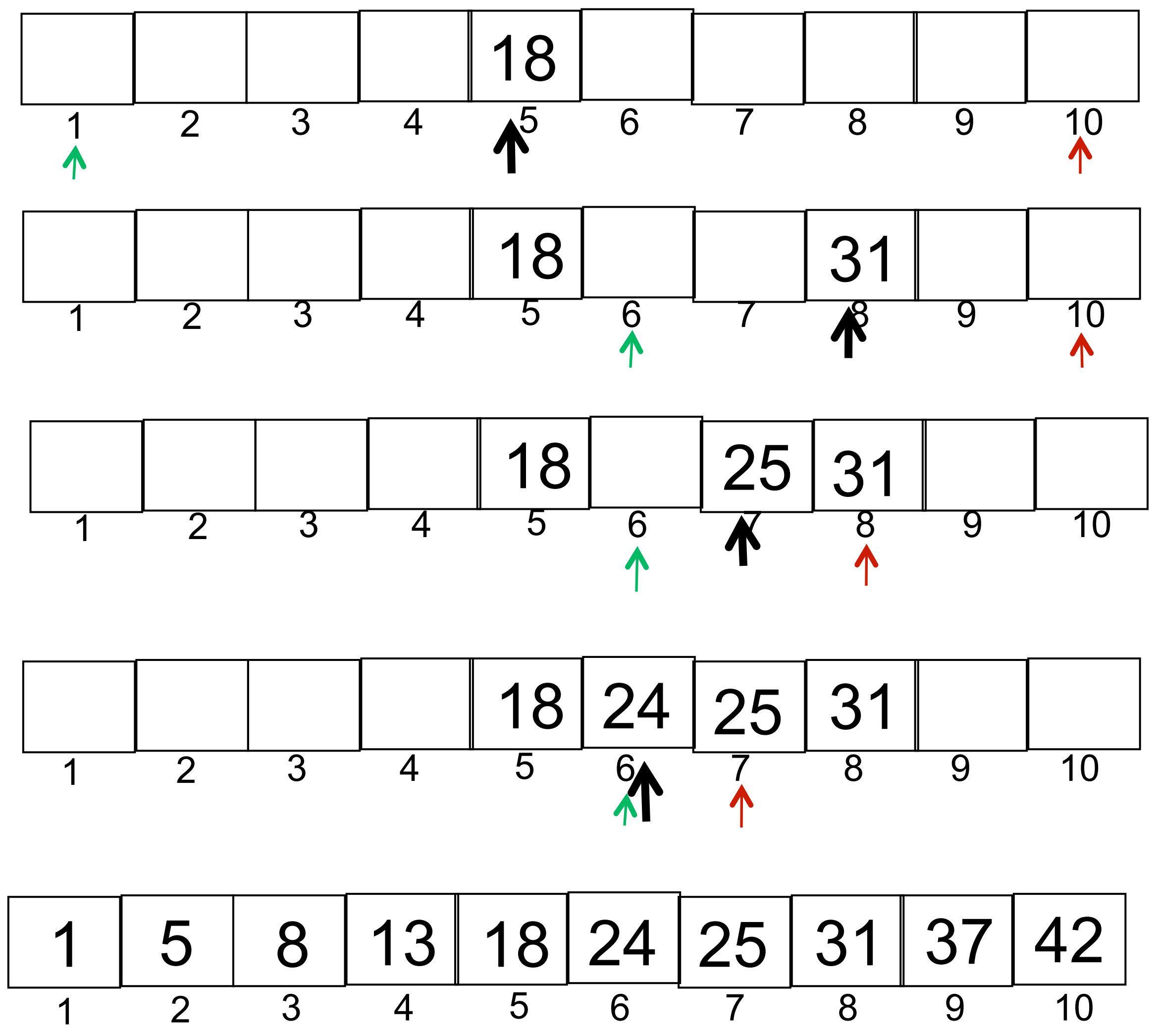
Sometimes, we know that the objects in the input sequence are given in some order. If so, we can do a different search that is better in the worst case.

We start by looking at the middle of the sequence. If that element is ordered before our desired object, then we shrink our view to be from the middle to the end, and keep looking in the same way; otherwise, we shrink our view to be from the start until the middle, and continue in the same way.

Eventually, our view will shrink to a single object. Either it is the one we want, or we know our object is not in the sequence.

Example binary search

Looking for the number 24 in the following list:



Binary search

Input: a finite sequence of objects *x[1], x[2], ..., x[n]*
Input: *y*, the object we are looking for
Output: *pos*, position of *y* in sequence, 0 if not there

```
1.  earliest := 1
2.  latest: = n
3.  while earliest < latest
4.      i := floor((earliest+latest)/2)
5.      if x[i] < y
6.      then earliest := i+1
7.      else latest := i
8.  if x[i] == y
9.  then pos := i
10. else pos := 0
11. return pos
```

Informal analysis of binary search

- does it terminate?
  - *Earliest* starts less than *latest*. Inside the loop, we set *i* to be between them. We then bring *earliest* up to *i*, or bring *latest* down to *i+1*, so the gap between *earliest* and *latest* is always shrinking. Eventually, they will be the same, and then the loop stops. So the algorithm terminates
- how many steps does it take?
  - at worst, if the object is not in the sequence, each time we halve the list. This means we do $\log_2 n$ times round the loop.

Next lecture ...

An Introduction to Graphs