REMINDER: Programs need to be:

* Correct: produce required results for valid inputs
* Reliable: behave sensibly for invalid inputs/ errors
* Efficient: give results quickly (even for large inputs)
* Maintainable: code is clear and well-structured

3 sorting functions for arrays:

* Random sort: choose random pairs of items, swap it out of order (RUBBISH)
* Bubble sort: repeatedly scan array, swapping out of order items (NOT GREAT)
* Quick sort: partition array into big / small, recursively sort partitions. (A LITTLE BETTER)
* You can compare performance of functions by time measurements.

IMPORTANT - Analysing and determining performance results:

* **Here is a set of results 🡪 Why did these results occur (in my opinion) 🡪 Based on that, what kind of algorithm might have produced those results.**
* You can mix things up
* **I’ve hypothesised that I’ve got this kind of algorithm 🡪 This algorithm should exhibit these kind of performance characteristics 🡪 It is exactly the same set of performance characteristics that I have observed when I test my program**

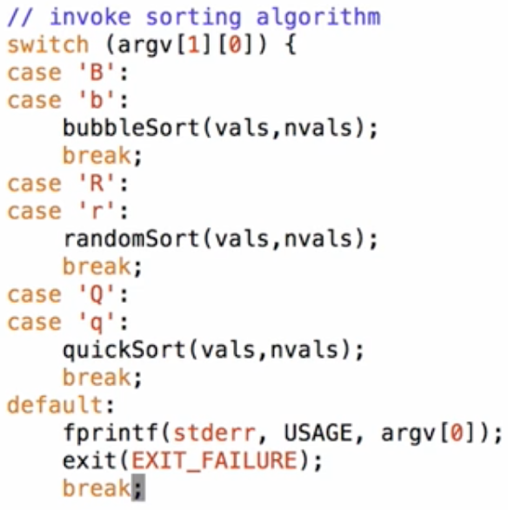
**Sort(a, n)** - What does SORTING mean?

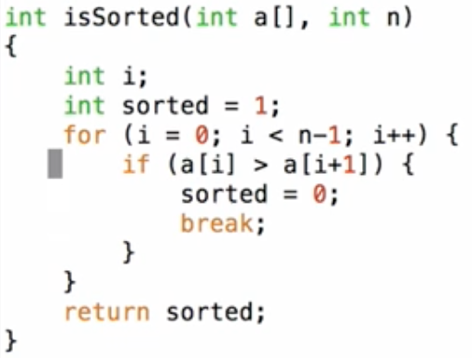
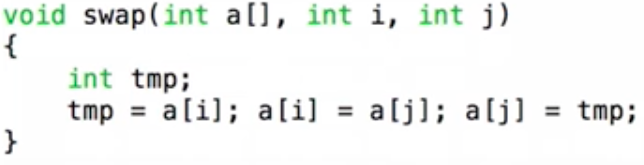
* PRECONDITION: Sorting is a function, where we sort an array a of n elements.
* POSTCONDITION: Forall i [0 … n – 1], a[i] <= a[i + 1]
  + Elements must be in ascending order
  + Duplicates are allowed.

./sorter X 5

./sorter X 6 < numbers

./sorter X ‘wc –l < numbers’ < numbers

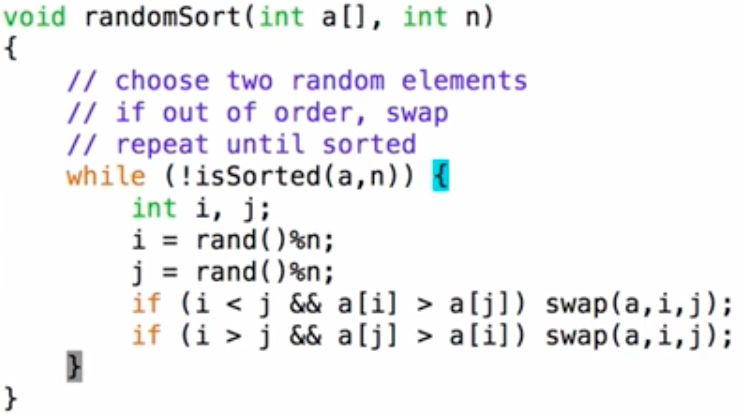




Implementing a SWAP function, which is usually needed to do sorting.

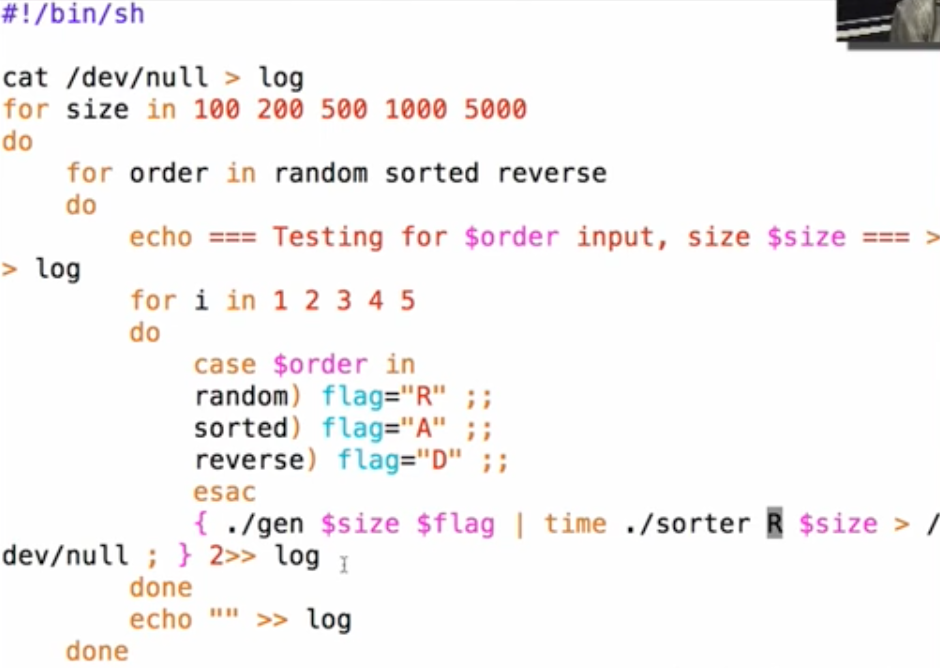
Determine if array is already sorted.  
Return 1 for yes, 0 for no.

randomSort Function



* After testing the randomSort function, the results are:
  + For an already sorted list, the randomSort function performs really well
  + For anything else (random, reverse etc.) the function performs poorly

Shell script to do automated testing



quicksort Function

* A lot faster than randomSort
* With a random generated input of 10,000 numbers i.e. “./gen R 10000”, randomSort takes forever to do, whereas quickSort is pretty much instant.

**Complexity Analysis / Sorting**

Complexity analysis isn’t trying to find an absolute cost measure, but it is trying to establish how fast the cost grows as you increase the size of input.

Allows us to understand performance of Algorithms.

* Define a function to characterised execution cost (time)
  + Identify value to measure size of input (N)  
    E.g. #items in data structure, length of input file, N in N! etc.
  + Identify core operation in algorithm
  + Express cost in terms of #operations = g(N)
* Shows how cost increases as input size increases
* Will the algorithm become infeasible for 100, 10000, 1000000, … ?
  + randomSort can only do about 10,000 random elements
  + quickSort can still do about 1,000,000 random elements
    - However, if you feed it reverse ordered / ascending ordered data, it will take longer.

Example: **Finding max value in an unsorted array**:

* Core operation = compare a[i] to MAX
  + This operation is guaranteed to occur at least once every single time the function is run
* How many times?
  + **N – 1 … O(n) (order n)**
* Execution cost grows **linearly** (i.e. 2 x #elements 🡪 2 x cost)

Example: **Finding max value in a sorted array (ascending):**

* Core operation = none
* No iteration needed, max value is ALWAYS last.
* How many times?
  + **Once … O(1) (order one)**
* Execution cost is **constant** (i.e. 9999 x #elements 🡪 same cost regardless of #elements)

Example: **Finding given value K in an array**:

* Core operation = compare a[i] to K
* How many times?
  + Not so straightforward…
* Need to consider best/worst/average-case costs.
  + Worst case: If K is not in the array, we need to scan the entire array before we discover that N is not there (result = -1 )
  + Best case:
* Edge case: value does not exist in the array, therefore return -1 (an index that is impossible to be in the array, rather than returning the index number i)