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**

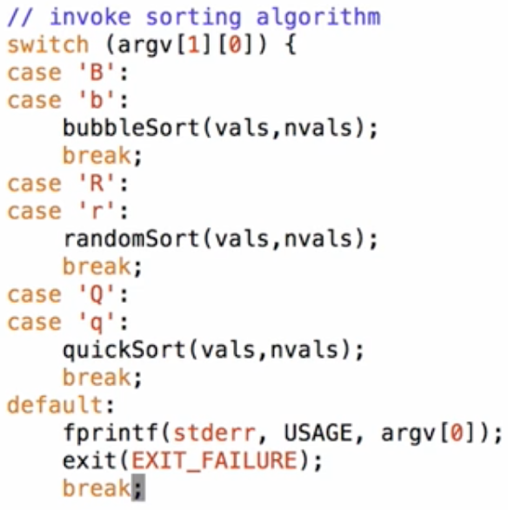
**Sort(a, n)** - What does SORTING mean?

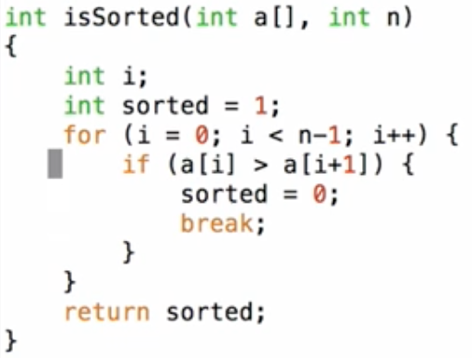
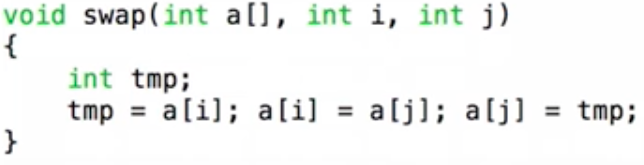
* PRECONDITION: Sorting is a function, where we sort an array a of n elements.
* POSTCONDITION: Forall i [0 … n – 1], a[i] <= a[i + 1]
  + Elements must be in ascending order
  + Duplicates are allowed.

./sorter X 5

./sorter X 6 < numbers

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

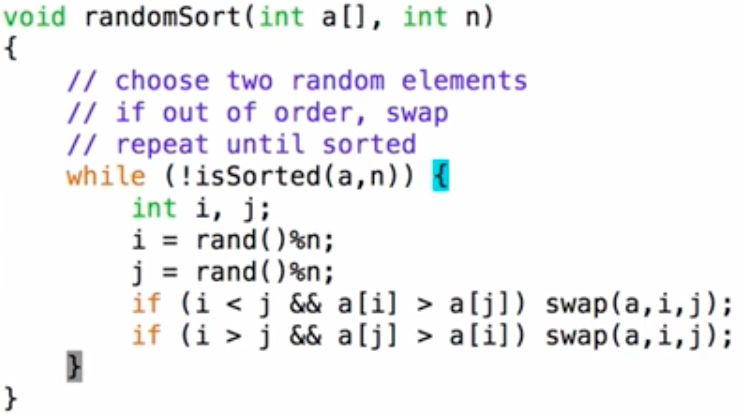




Implementing a SWAP function, which is usually needed to do sorting.

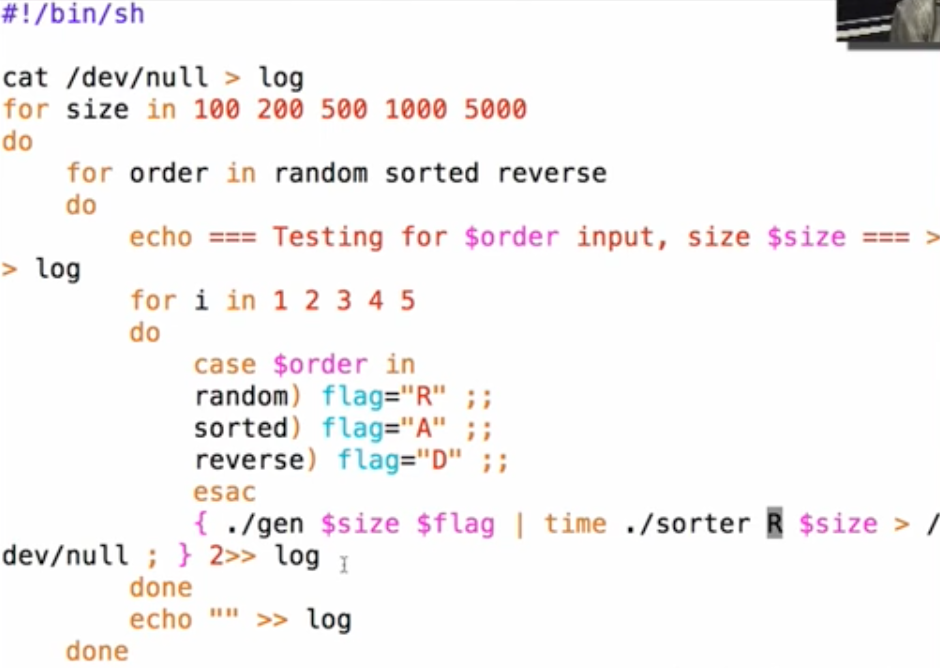
Determine if array is already sorted.  
Return 1 for yes, 0 for no.

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 (comparison operation)
  + 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” (index into array)
* 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…
* 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)
* 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 )  
    **(cost = N)**
  + Best case: If K is the 1st element of the array.  
    **(cost = 1)**
  + Average case: If K is somewhere in the middle of the array.  
    **(assuming all elts in array are equally searchable cost = N/2)**
* Analysis:
  + Best case = 1
  + Worst AND Average case = O(n), because if you double the size of the array, it still takes twice the time to reach the middle of the array to find K
  + In “absolute” terms, average case is cheaper but the important thing is that they both **GROW AT THE SAME RATE**

**Algorithmic Complexity**

Cost is primarily interest for large data sets.

* Consider the **growth rate** rather than **absolute cost**

Leads to **Complexity Classes** and **Big-O-Notation**

* O(1) … constant functions
  + g(n) = 1, 2, 3 …
* O(logn) … log functions
  + Log(n), log2(n), …
* O(n) … linear functions
  + n/2, n, 5n+100, …
* O(nlogn)
  + O(n2), O(n3), O(kn), O(n!), …
  + O(n!) = n factorial.  
    E.g. 10 factorial = 109. 100 factorial = 10099.

Algorithms are “rated” by their complexity class

* Therefore, we might say that “quicksort has worst case complexity O(n2)”

Assigning an algorithm to a complexity class

* Determine cost g(n) as function of input size n
* Associate g(n) with appropriate complexity class

<http://bigocheatsheet.com/>

**EXERCISE: Assigning Complexity Class**

Reminder:

* Core = core operation of the algorithm
* G(n) = number of iterations
* Complexity = growth in cost
* With **order notation**, we always **throw out all the constants** and pick the **highest order term**
  + E.g. if our cost function was x3 + 2x2 + 10000x – 5  
    We would quote it as O(x3)
  + The smaller exponents would matter less and less as input size grows bigger
  + We are abstracting away as much as possible, so we know how fast the cost will grow

**STRUCTURE 1: Finding max value in sorted (ascending) linked list**

This linked list structure has:

* Pointer to head of the list
* Pointer to current node of the list

Therefore:

* Core: Comparison – greater than comparison (>)
* G(n): n times
* Complexity: O(n) – Linear

**STRUCTURE 2: Finding max value in sorted (ascending) linked list**

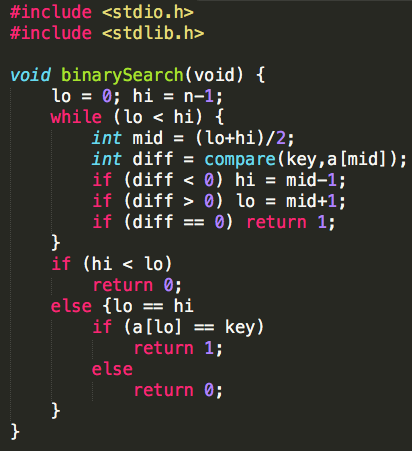
This linked list structure has:

* Pointer to head of the list
* Pointer to tail of the list
* Pointer to current node of the list

Therefore:

* Core: Deferencing ( -> )
* G(n): 2 times
  + Deference twice, because we go from L->tail then tail->value
* Complexity: O(1)
  + Order 1 / constant, because no matter how large the list is, we can always go straight to the end of the list because of the tail pointer.

**Differences in the data structure can heavily affect complexity of an algorithm**



**Search for value in sorted array** **(binary search)**

1. Pick a midpoint
2. Is the value less than or greater than midpoint
3. Repeat process until value is found.

* Low point keeps track of lower boundary
* High point keeps track of higher boundary
* If Low == High, then that is the key.
* Low can cross high

Therefore:

* Core: Comparison
  + (<) lesser than comparison
  + (>) greater than comparison
  + (==) equal comparison
* G(n): log2n times
  + Start with array of certain size, split it in half,  
    choose one half, split it again and repeat.
* Complexity: O(logn)

**Sorting**

Why sort?

* Makes searching easier (binary search vs linear search)
* Useful for reading reports / lists / tables

Why study sorting?

* Easy to describe the problem
* A variety of approaches exist (you can do performance comparison)

Sorting on Linux

* **Sort** command
  + Sorts a file of text, understands fields in line
  + Can sort alphabetically, numerically, reverse, random
* **Qsort** function
  + **qsort(void \*a, int n, int size, int (\*cmp) () )**
  + Sorts any kind of array (n objects, each of size bytes)
  + Requires the user to supply a comparison function (e.g. **strcmp()**)
  + Sorts lists of items using the order given by **cmp()**(pointer to comparison function that quicksort uses)