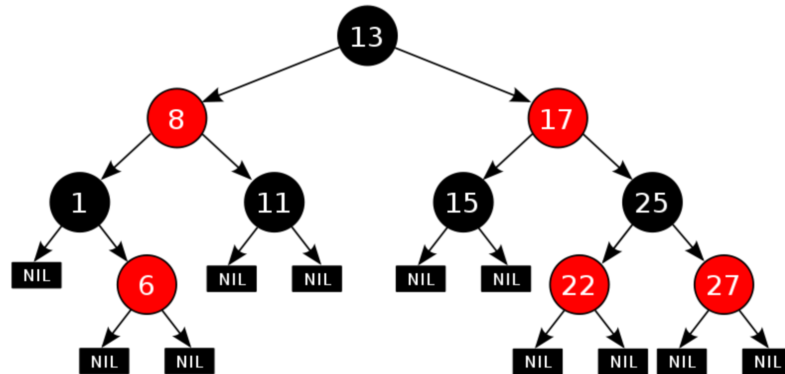
Algorithms I
Dr Robert Harle



CST Paper I

(IA NST CS, PPS CS and CST)

Easter 2009/10

OOP Feedback

"Audience participation was good in retrospect but felt like cruel punishment at the time"

OOP Feedback

"...the last lecture felt like a complete waste of time."

"I did very much enjoy the last session!"

OOP Feedback

"the 5 minute breaks were excellent"

"...we wasted time in the middle of the lectures"

OOP Feedback

"Needs more Blobfish"

"More Blobfish"

"I like Blobfish"

"Blobfish rules!"

"Blobfish Blobfish Blobfish Blobfish…"

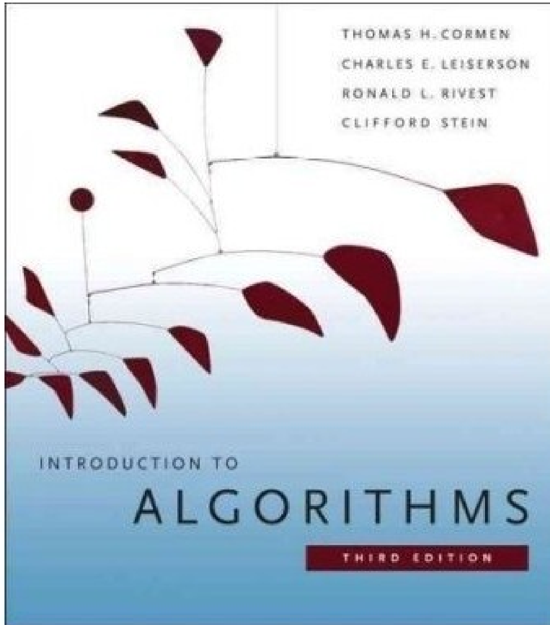
Algorithms I

- This course was developed by **Dr Frank Stajano**, who is on sabbatical this year
- I'm the "substitute teacher" :-)

- Dr Stajano's notes are very good: you have a copy of those as the handout. Those and the course textbook are probably all you need.
- However, I will post an annotated PDF of the notes I make in lectures as we go: check the course web page

- Three Parts
  - Sorting Algorithms
  - Algorithm Design
  - Data Structures

The CLR(S) Book



- Intro. To Algorithms
  - Cormen, Lieverson, Rivest, (Stein)
- The course is loosely based on this book
  - Definitely read the relevant bits of this book
  - Most libraries should have a copy
  - It contains some good exercises

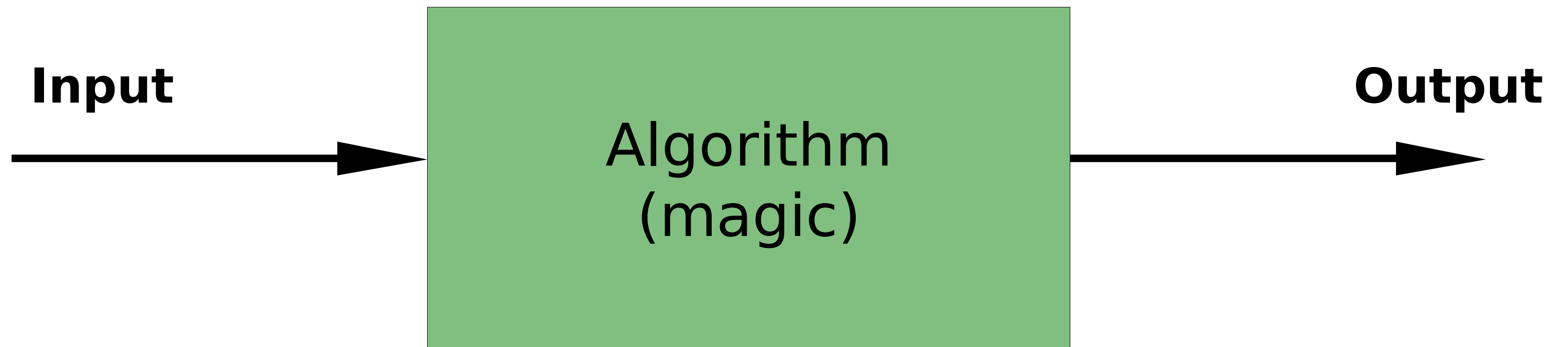
Exercises

- There are some exercises dispersed throughout the notes
  - They aren't numbered
  - Most are just meant to be done as you read, rather than detailed problems
- There will be an exercise sheet available as a PDF on the course website that you may wish to use for supervisions.

Algorithms

- At its core, CS is really just about puzzle solving. But we aren't just interested in finding a solution (or "algorithm"), we're interested in finding the best solution given some definition of 'best'
- Everything else (programming, maths) is just a set of tools that turn out to be useful in supporting our puzzle solving.
- There is no "universal algorithm"; nor will there be.
  - But you can learn a lot from studying how to solve a variety of problems since many problems can be broken down into smaller problems to which established algorithms (or variants of) are appropriate

Algorithms Optimize Something



- We choose algorithms based on:
  - How soon they give us output (performance)
  - How much resource they use (space)
  - How good the output is (quality)
  - Combinations of the above

Example: Digital Cameras (JPEG)

- Digital cameras read in a load of pixels and have to convert them into a JPEG image
  - **Performance**: Need to do the conversion quickly so you can take another picture
  - **Space**: Need to do the conversion with minimal space overheads (to keep camera cost and size down)
  - **Quality**: Need to produce a small file that is still a good representation of the original data

Example: Search Engines

Pages: A B C D E F G H I J K L

Index							
GET	A	B	F	H			
A	G	D	K	I	J	B	D
FIRST	G	A					
THIS	E	F	I	G	A		
YEAR	C						

- Algorithms:
  - Look up the search term in the index
  - Optionally combine the results (AND, OR)
  - Arrange the results in some useful order

Part I: Sorting Algorithms

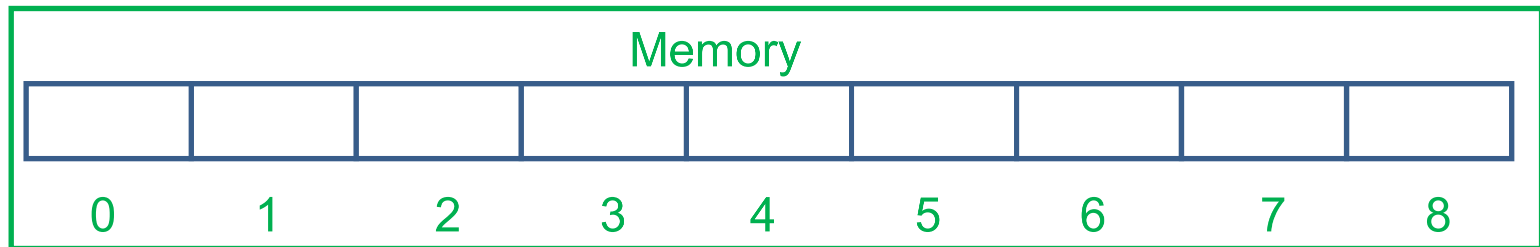
Why Sorting?

- There is an objective correct result
- Many sorting algorithms are available
  - Some really simple
  - Some more complex
- Sorting (and searching) are needed for most large-scale algorithms

- You have already met some of this in FoCS, but I'll recap anyway (it is revision time after all)
  - Plus you concentrated on sorting **lists** in FoCS: here we look at sorting **arrays**

Memory Model

- We'll use the simple model from OOP



- Key points:
  - Memory is addressed using numerical addresses and therefore random access
  - We will assume that we never run out of memory
  - We will not worry about the capacity of each memory slot (we'll assume any number can be represented in any slot)

Insertion Sort

Insertion Sort

```
0   def insertSort(a):
1       '''BEHAVIOUR: Run the insertsort algorithm on the integer
2       array a, sorting it in place.
3
4       PRECONDITION: array a contains len(a) integer values.
5
6       POSTCONDITION: array a contains the same integer values as before,
7       but now they are sorted in ascending order.'''
8
9       for k from 0 to len(a)-2:
10          assert(the first k positions are already sorted)
11
12          # Pick up item k+1 (call it a[j]) and let it sink to its correct place
13          j = k+1
14          while j > 0 and a[j-1] > a[j]:
15              swap(a[j-1], a[j])
16              j = j-1
```

How 'good' is any algorithm?

- It's hard to put numbers to anything since the performance is presumably heavily dependent on the input

- As you know we usually study the limiting behaviour using the **asymptotic notation** you met in FoCS

Complexity Notations

Big-O: $0 \le f(n) \le k.g(n)$

Θ: $0 \le k_1.g(n) \le f(n) \le k_2.g(n)$

Ω: $0 \le k.g(n) \le f(n)$

For $n>N$

$K, k_1, k_2, N > 0$

Notes

- $\log_a(x) = \log_b(x)/\log_b(a)$
  - So the base of any logarithm in g(n) is irrelevant
- The value of N above which the bound holds could be very big
  - i.e. Take care when comparing two complexities for small n.

Examples

- Show $(x+5)\lg(3x^2+7)$ is $O(x\lg x)$

Examples

- Show n^3+20n is $\Omega(n^2)$

Examples

- Show n^2-3n is $\Theta(n^2)$

Relating to Running Time

- We assume:
  - Any memory access takes unit time
  - Any arithmetic takes unit time
- Thus the running time is linked to the number of operations the algorithm requires.
- Problem: this is often dependent on the input

Worst, Average and Amortized costs

- **Worst-case**
  - Analyse for the worst possible input. This gives you an upper bound for the performance.
- **Average-case**
  - Analyse for an 'average' input. Problem here is that the notion of average assumes some probability distribution of inputs, which we rarely have (and which is application specific of course).
- **Amortized analysis**
  - Sometimes we have a sequence of operations that occur: in this case we may *amortize* the total cost to run the sequence of operations so we get an average cost per operation. e.g. Garbage collection.

Insertion Sort Analysis

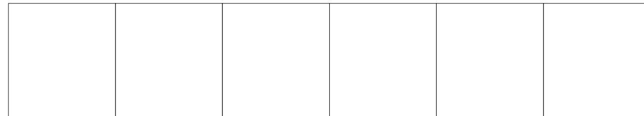
```
 9  for k from 0 to len(a)-2:
10      assert(the first k positions are already sorted)
11
12      # Pick up item k+1 (call it a[j]) and let it sink
13      j = k+1
14      while j > 0 and a[j-1] > a[j]:
15          swap(a[j-1], a[j])
16          j = j-1
```

Is that Optimal?

$O(n^2)$ Sorting Algorithms

- There are lots of these around. They tend to be easy-ish to think up. Be careful not to reinvent the wheel!
- We're going to look at a few more so that you have been exposed to the common ones and you also get more practice analysing complexity.
- You will find a lot of CS types scoff at these algorithms as being a waste of time. But they have definite advantages:
  - They are usually concise and understandable (means fewer implementation bugs)
  - For small n, they might be the quickest method!

Selection Sort: Idea

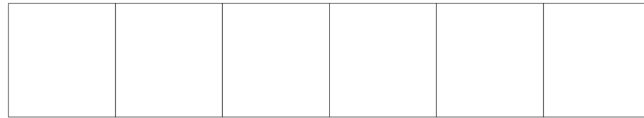


Selection Sort: Example

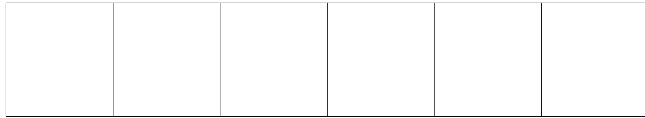
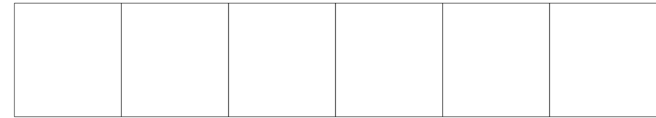
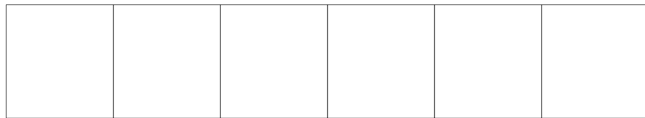
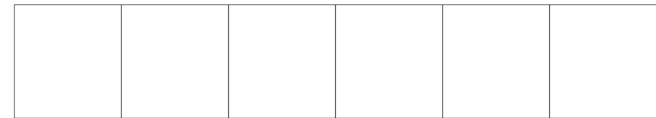
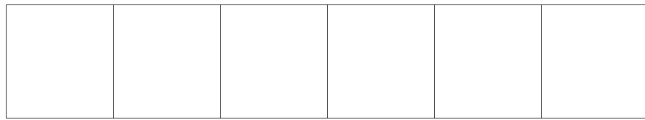
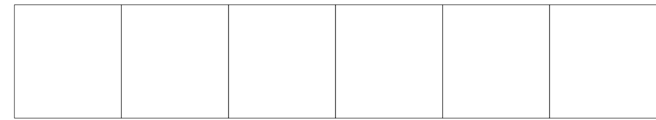
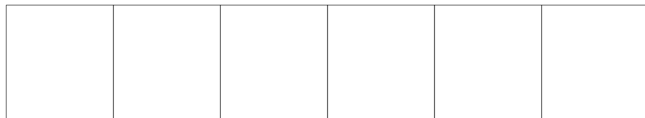
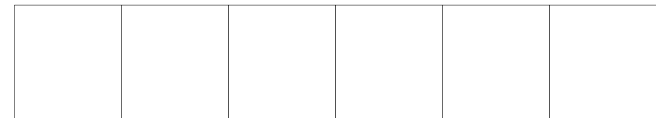
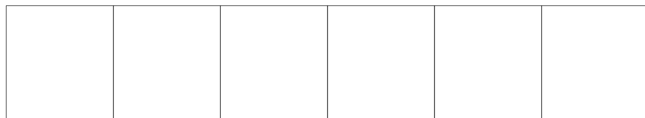
Selection Sort

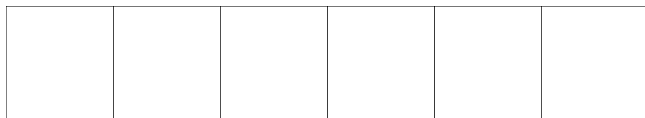
```
 9  for k from 0 to len(a)-1:
10      assert(the positions before a[k] are already sorted)
11
12      # Find the smallest element in a[k..end] and swap it into a[k]
13      iMin = k
14      for j from iMin+1 to len(a)-1:
15          if a[j] < a[iMin]:
16              iMin = j
17      swap(a[k], a[iMin])
```

Binary Insertion Sort: Idea



Binary Insertion Sort: Example

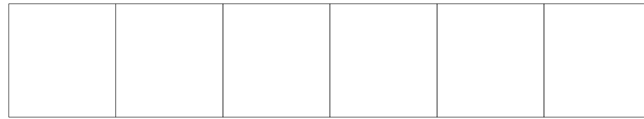












Binary Insertion Sort: Analysis

Binary Insertion Sort: Analysis

Bubble Sort: Idea



Bubble Sort: Example

Bubble Sort

```
9   repeat
10      # Go through all the elements once, swapping any that are out of order
11      didSomeSwapsInThisPass = false
12      for k from 0 to len(a)-2:
13          if a[k] > a[k+1]:
14              swap(a[k], a[k+1])
15              didSomeSwapsInThisPass = true
16  until didSomeSwapsInThisPass == false
```

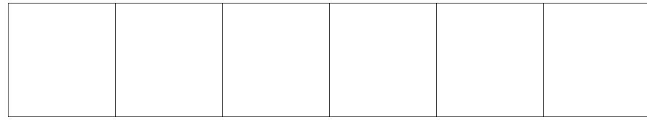
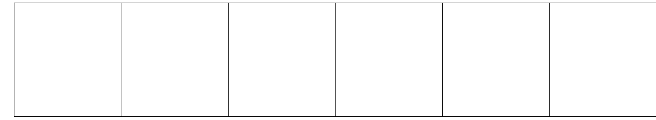
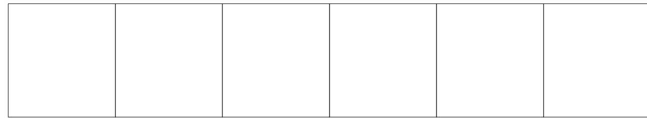
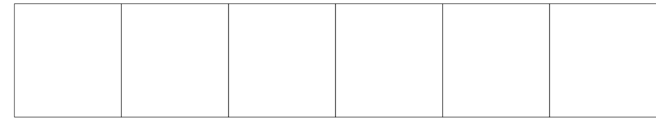
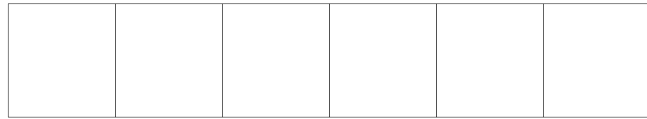
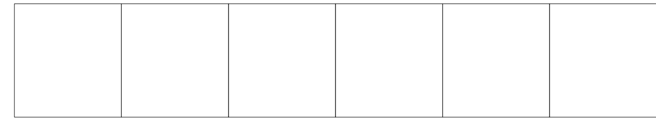
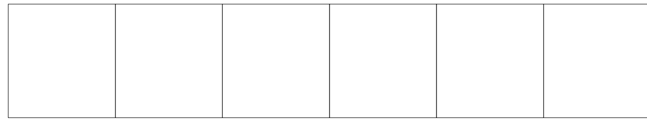
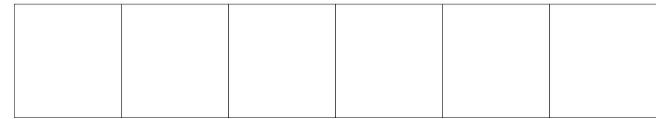
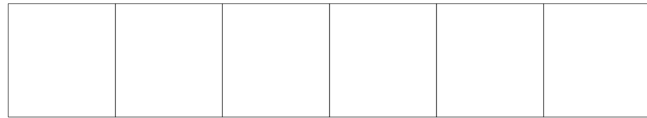
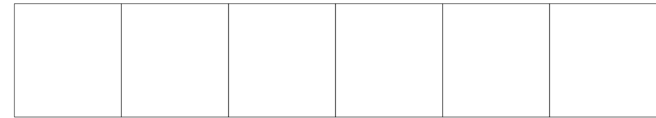
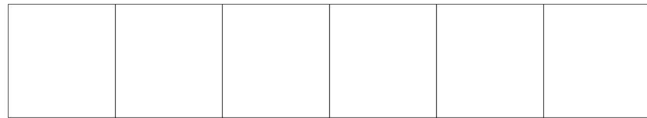
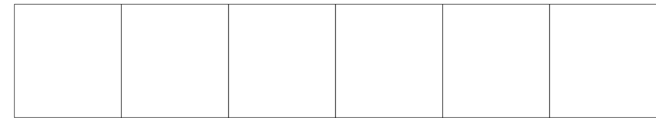
Better Algorithms

- Can we actually hope to get close to our theoretical O(nlogn) limit?

- You know we can because you've met mergesort before.
  - In ML it was a lovely short algorithm that did a great job sorting lists
  - It has some drawbacks when we look at arrays but we'll get to that shortly...

Mergesort: Idea (recap)

Mergesort: Array Example

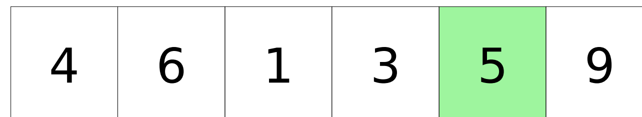













Mergesort: Analysis

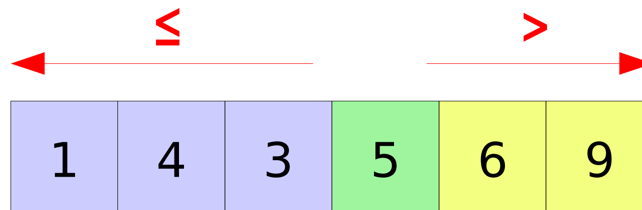
Mergesort: Analysis

Quicksort: Idea

- Recursive like mergesort, except it doesn't just slice the array into two
- The basic idea is to pick a pivot element
  - Any element will do, although we might get better results if we choose more carefully

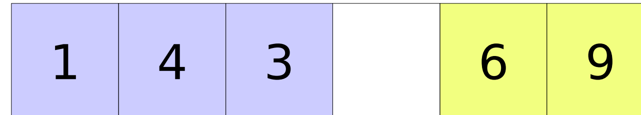


- We then partition the array into those bigger than the pivot and those smaller than the pivot

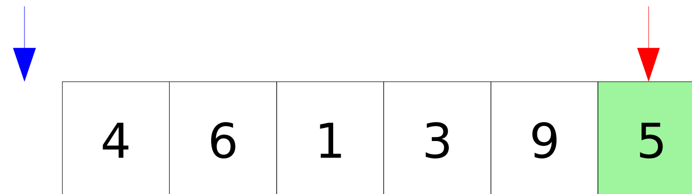


Quicksort: Idea

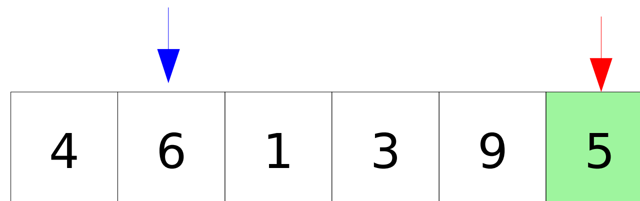
- Now we recursively apply quicksort to the two partitions



- How do we partition?
  - Have two pointers, L and R

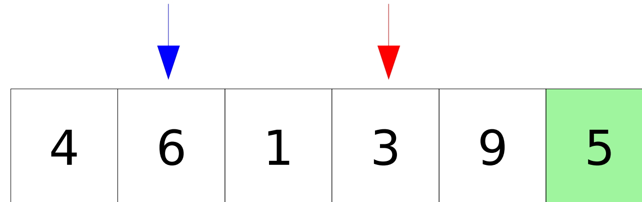


1. Increment L until it points to something bigger than the pivot or L==R

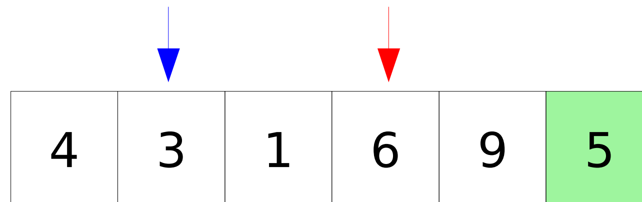


Quicksort: Idea

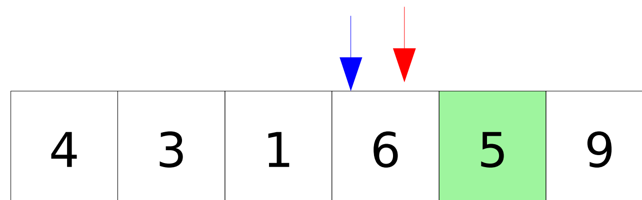
2. Decrement R until it points to something smaller than the pivot or L==R



4. If (L != R) then swap the elements

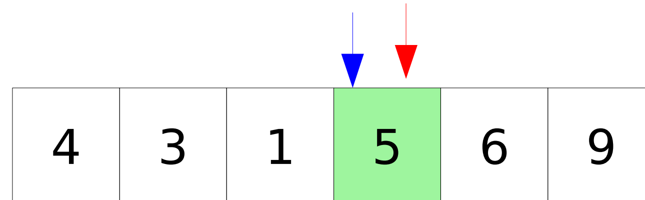


6. If (L != R) go to step 1 else continue



Quicksort: Idea

5. Swap the pivot into the position R



- Partitioning done!

Beware: Depending on where your pivot is chosen to be, you need to think carefully about what gets swapped where: it's very easy to end up being off by one. You will find lots of subtle variations on the algorithm implementations (all of which work). I recommend you try writing a quicksort that works on any pivot (the one here works just if you choose the last element as the pivot)

Quicksort: Example

Quicksort: Example

Quicksort: Example

Quicksort: Analysis (Gulp!)

- Best case
  - Magically we manage to always choose the pivot that splits the subarray being sorted into two
  - Then we have exactly what we had for mergesort
  - $f(n) = 2f(n/2) + kn$
  - That was O(nlogn) :-)

Quicksort: Analysis (Gulp!)

- Worst case
  - We somehow manage to choose the biggest (or smallest) element as the pivot every time
  - Now we will always recurse on a subarray of size (n-1)
  - $f(n) = f(n-1) + kn$
  - This is $O(n^2)$:-)

Quicksort: Analysis (Gulp!)

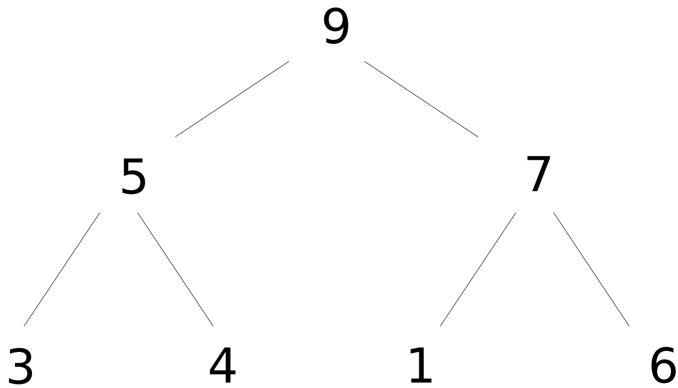
- So why is quicksort **the** choice for sorting?
  - It has a poor worst case
  - But the best case is generally *better* than mergesort (smaller constants in O(nlogn)) and the average case is still O(nlogn)
  - And quicksort sorts in place
  - So it comes down to trying to avoid the worst case, which is all about pivot choice

Order Statistics with Quicksort

Heapsort

- One last interesting algorithm
- Sorts in place and guarantees O(nlogn) for any input!
  - Although the constant of proportionality is greater than for quicksort
- Particularly interesting for us because it is based on a data structure called a heap (which we will use later on too)

Introducing Heaps

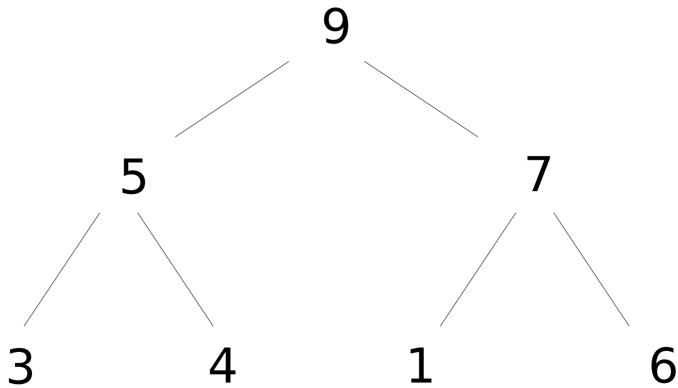


- Consider a simple binary tree (two branches out of each node)
- There is only one rule: the value of the two children must each be less than the value of the parent

Note:

- The root of the tree is always the biggest number
- The root of any subtree is the biggest number in that subtree

Introducing Heaps



- Now we represent this tree using an array in a certain way:
- The children of the node at [i] can be found at [2i+1] and [2i+2] (array starts at zero)

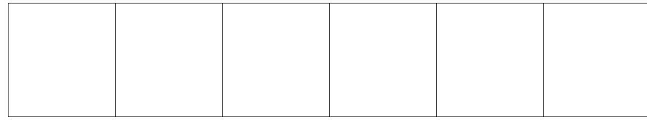
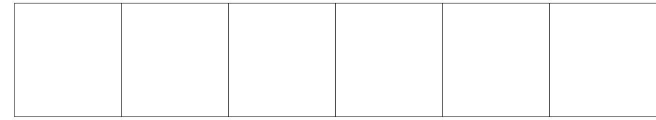
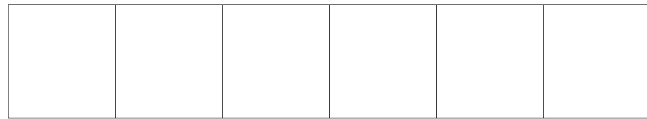
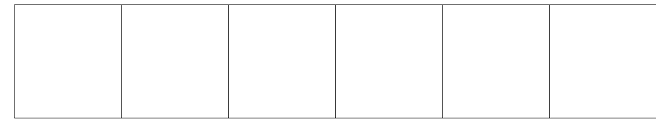
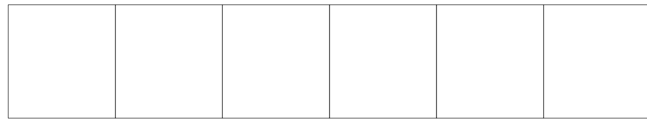
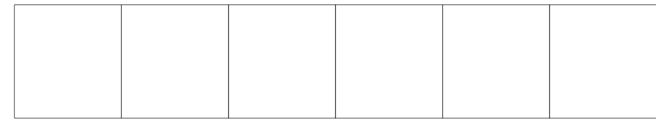
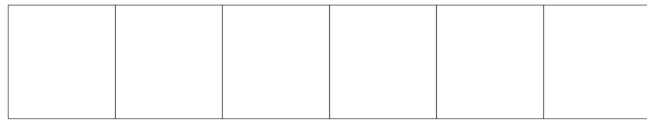
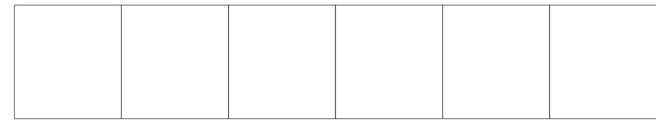
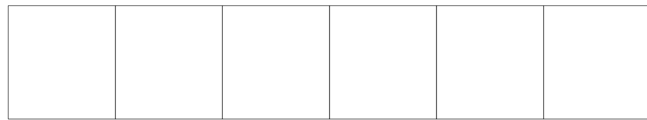
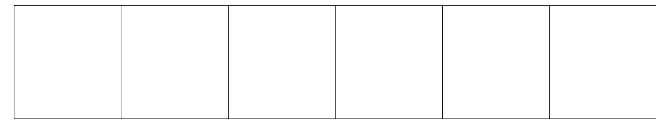
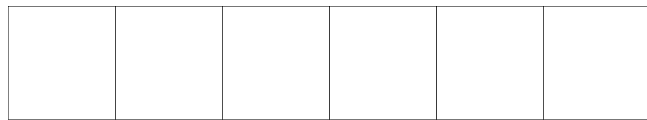
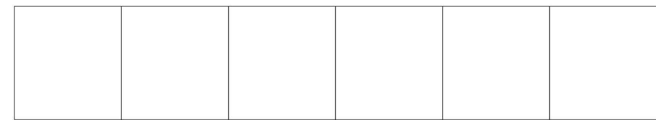
0	1	2	3	4	5	6
9	5	7	3	4	1	6

Heapsort

- Heapsort proceeds as follows:
  1. Make your data into a heap in the array [0...n]
  2. Swap the element at the end of the array with the head of the heap
  3. Heapsort the array [0...n-1]
  4. Loop to step 2.

Heapsort: Graph View

Heapsort: Array View

Heapsort: Analysis

Heapsort: Analysis