**Disclaimer. Please do not copy these exactly, I made a mistake he commented on saying that many of us have the same mistake. As this is a scored module based on notes, copy pasting these exactly as they are is against the uni guidelines yada yada. You know the drill.**

**READ NOTES IN ORANGE**

**Monday 14th March 2016**

**March 14th**

Today we are going to be looking at the implimention of partitioning.  
  
Partitioning.   
  
In remembering we start looking for incorrectly placed items left and right of the pivot element and swap if found.

In **Setting**work in int keys[size]; pivot gives index of pivot element have initially left/right denote leftmost/rightmost index

It is guaranteed to stop because we are left of the pivot and we are moving to right. after we then swop using a a standard swop, then we check if the pivot element was swapped.

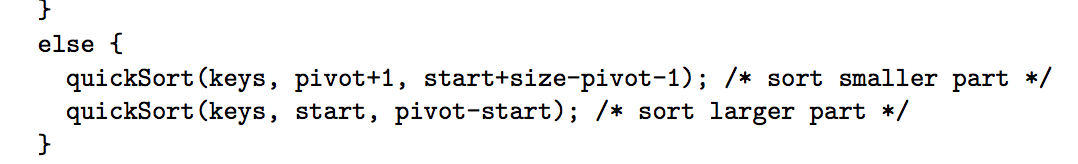
**Quick Sort.**

There are two ways to do this, a bad way and a better way.

To understand this, We need to look at the visual below.   
  
We recursively sort the left and the right, it is a relatively simple algoriythm. With quick sort. The main work lies in partition (plus recursion) • choice of pivot element completely arbitrary. It is also important to note that. recursion death depends on the size of the different parts.

**Better Quick Sort.**

If your compiler is very good. instead of putting everything on the call stack, you can overwrite the parameters with the last two lines seen below. This is screenshot of the last two lines in question.



The main work lies in parition (plus recursion), the  choice of pivot element completely arbitrary, recursion depth depends on the size of the smaller part if the compiler is clever enough to handle second recursion as iteration (know as tail call optimization)

The important thing to remember is **Let n = size, T(n) total run time** The observations are **T(n) = Θ(n) + T(s) + T(n − s − 1)** where s depends on position of pivot after partition

We have no idea what S is but the runtime depends on S. in extreme cases S is always 1. The result is the sum of a linear number of numbers.

extreme case s = 1 always T(n) = Θ(n) + Θ(1) + T(n − 2)

In the worst case scenario the runtime of quick search is a large as the runtime of search and sort.

In the best case it is really quick and in worst case it is super slow this is beacause.

A bad algorithm will take a long time to run, **quick sort is slow!**

The question must be asked, can we improve the worst case and is it possible to avoid having bad luck all the time. I assume yes but lets see the facts.  
  
One can have back luck once and twice or even a couple of times but in the long run things will eventually even out.

**Avoiding Bad Luck**

**Fact** one can have bad luck once or twice or even a couple of times but in the long run things will ‘even out’

**Idea** select pivot element randomly

This is different from choosing a fixed position because randomising the selection of the pivot moves your reliance of ‘bad luck avoidance’ from input to the random choices made.

This helps  because input can have structure structure of input may be precisely bad for algorithm random choices have no structure so having bad luck often is very unlikely.

**Extra information.**  
  
I have told Dr J that I have shared a vid to insert here for the class so don’t worry about inserting it . He was happy with the last one

Insert video

**Selection Problem**

Input array long keys[size] and int r Output position int p such that keys[p] is r th smallest item in keys

**Examples** r = 1 yields minimum, r = size yields maximum

                   r = size/2 yields median, r = size/4 yields lower quartile

From the example above, we already know how to solve this because we already know quick sort.

**Insert pic here**

In the pic above we did a comparison of the types of search and observed which was faster, at one point the fastest overtook the slower and we were able to see proof of what was discussed in the lecture.

The key is randomised quick sort.

We pick our pivot element randomly then we crawl the partition.

**Selection Problem**

Input array long keys[size] and int r Output position int p such that keys[p] is r th smallest item in keys

**Examples** r = 1 yields minimum, r = size yields maximum

                   r = size/2 yields median, r = size/4 yields lower quartile

From the example above, we already know how to solve this because we already know quick sort.

Insert SCreenshotted pic seen in group chat

In the pic above we did a comparison of the types of search and observed which was faster, at one point the fastest overtook the slower and we were able to see proof of what was discussed in the lecture.

The key is randomised quick sort.

We pick our pivot element randomly then we crawl the partition.   
  
This essentially solves the search problem. The demostrations on the screen (Screenshotted above clearly showed the winner.