Contents

[Analysis 1](#_Toc527147404)

[Background 1](#_Toc527147405)

[Independent Feedback 3](#_Toc527147406)

[Systems used 3](#_Toc527147407)

[Objectives 4](#_Toc527147408)

[Design 5](#_Toc527147409)

[Class Diagrams from solution: 5](#_Toc527147410)

[The design Flowchart of the Simulation: 7](#_Toc527147411)

[Methods/Attribute description table: 8](#_Toc527147412)

[Sorting Algorithm for quarklist/partIdentify – Merge Sort 11](#_Toc527147413)

[NEA Rough Notes: 12](#_Toc527147414)

[Possible edits to NEA 14](#_Toc527147415)

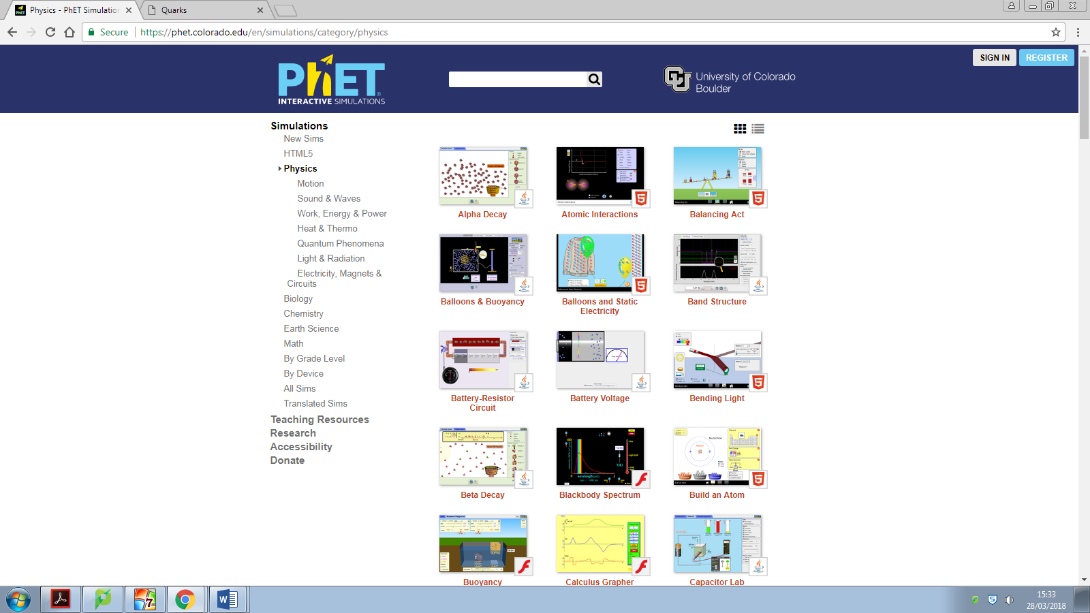
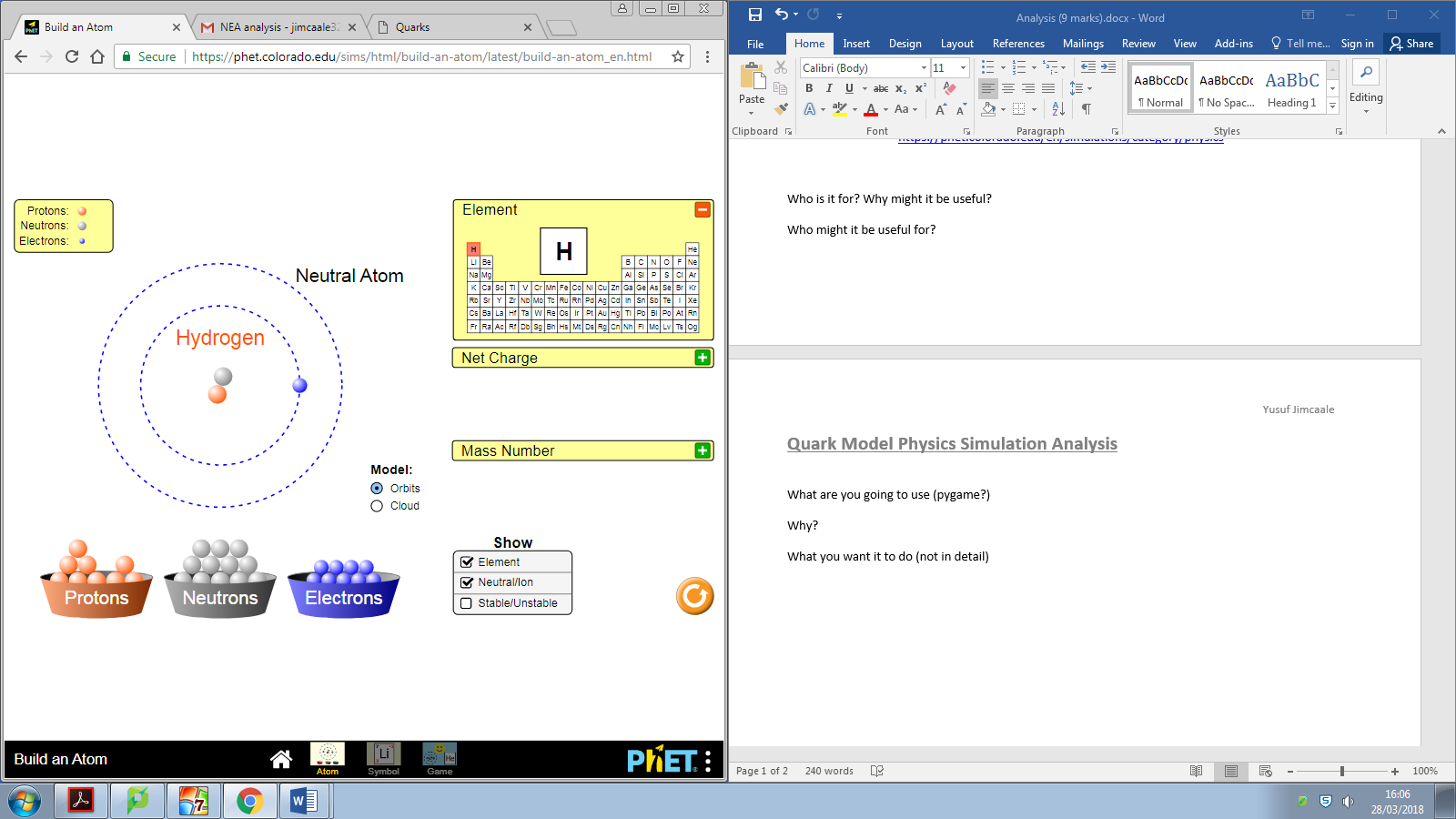
# Analysis

## Background

Throughout the course of my Physics lessons, my teachers often used computer-generated simulations to help explain topics with ease. The visual nature of the simulations (in my opinion) allowed for students (such as myself) to gain a more solidified understanding of the area in question. After several discussions with one of my physics teachers, I ended up finding the website from which he got many of these simulations – PhET Lab Simulations. After trying a couple of these simulations, I came across one, which really intrigued me – “Build an Atom”. The goal of this simulation was to show students how changing the number of protons and neutrons could also change an element and thus its properties. The program’s use of mouse interactions to press buttons and move around particles was something I really wanted to attempt in replicating (programmed in HTML 5). The website had a wide variety of simulations – which meant I was not short on ideas on what to do for my project. However, despite its vast wealth of resources, there were still some topics, which did not have simulations specifically for them.

Whilst being taught about quarks and the Standard Model, I realised that there were no simulations at all for it – and given the nature of the topic, many of my classmates found themselves unsure and confused. After seeing this, I came up with the idea to create a simulation program, which would allow users to mix and match quarks together and allow them to pick correct configurations, to successfully create subatomic/composite particles. The program’s user interface will take inspiration from the “Build an Atom” simulation – with the solution including a particle space, quark buttons and a place where information about the particle is outputted. In many ways, my project aims to adapt this “PhET Simulation” and apply it to a very different concept.

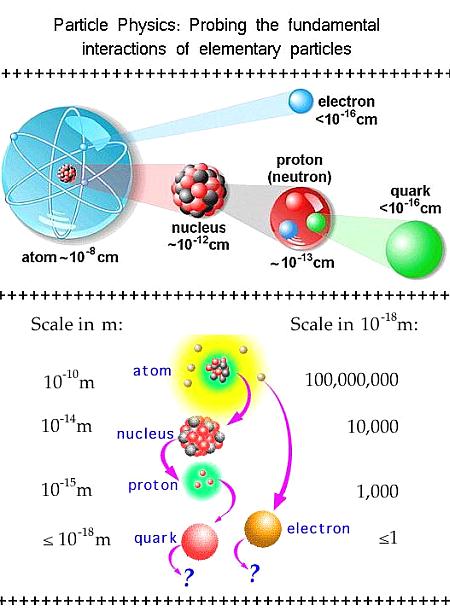
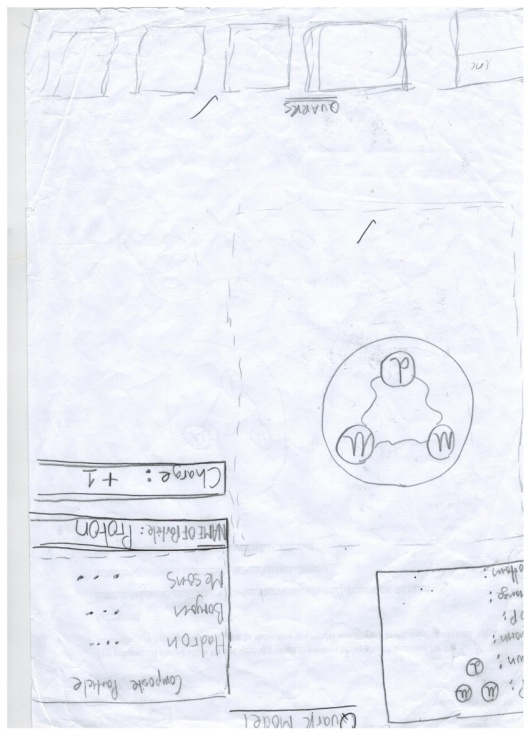
Whilst the simulation is running, the user will be able to press labelled buttons which then generate colour coded quarks onto the particle space; the quark tally will also be incremented by one (depending on what quark button was pressed). They will be able to do this until the particle space becomes full, by which the user will be given a message letting them know of this. If the quarks in the particle space match a configuration of that of a type of particle (i.e. “up”,”up”, “down” for a proton), then the particle’s name and information will be outputted (stored in a class/read from a text file). Composite particles can either contain two or three quarks depending on their configuration. Twelve quark buttons will be at the disposal of the user – which includes six quarks and six anti-quarks (the antimatter version of quarks), with a combination of 12 possible particles to create – with each one having its information such as name, charge and type being stored beforehand. In addition to these 12 buttons, there will be a “REFRESH” button which wipes out the contents in the particle space, particle information and the quark tally – enabling the user to start with a clean slate and help facilitate experimentation with different combination types.



**Screenshot of “Build an Atom” Simulation** - <https://phet.colorado.edu/sims/html/build-an-atom/latest/build-an-atom_en.html>

**Screenshot of PhET Physics Simulations** - <https://phet.colorado.edu/en/simulations/category/physics>

Quarks are a type of elementary particle and a fundamental constituent of matter. Quarks combine to form composite particles called hadrons, the most stable of which are protons and neutrons – which make up the nucleus of an atom. There are six types of quarks, known as *flavours*: up, down, strange, charm, top, and bottom.  For every quark flavour there is a corresponding type of antiparticle, known as an anti-quark, that differs from the quark only in that some of its properties have equal magnitude but opposite sign. Quarks are also the only known particles to possess electric charges which are not integer multiples of the elementary charge “e”.



## 

Particle Diagram showing how quarks make up composite matter <https://sites.google.com/site/csetstudyguidechemistry/home/1-2c---atomic-structure>

## 

My preliminary sketch of my proposed solution

## Independent Feedback

The solution to this project is to enable for the topic of Fundamental particles to be represented in the form of a simulation - so that students can more easily understand the topic. As a student myself studying physics currently, this project will allow me to combine both my physics knowledge and my coding/computer science skills into creating a simulation. After proposing this idea to my physics teachers – they gave me advice on how to make the simulation work effectively as a teaching tool – including extra information about the particles; adding extra “facts” which work to add a bit of entertaining value for the students.

## Systems used

In terms of programing language, I am planning on using python to code my solution; having used it for a couple of years now, I will not have to spend any extra time learning a new coding language. In addition, the scale of my project will mean that my final solution will be perfectly suited to it. Upon looking for python libraries, I came across pygame. This library seems to be the most appropriate for my solution as it is able to deal with user inputs as well as creating a game window from which the simulation could be played.

## Objectives

1. Create GUI
   1. Allow user to click on buttons that add quarks to particle
      1. Create UI that has space for Quark picture, user info, and buttons for the potential quarks
   2. Present information to screen about particle when the quarks represent a particle
      1. Get user input that will add quarks
      2. Check particle representation against actual particle make ups
      3. Load information from file and produce on the screen and load nothing when relevant
   3. Allow user to start again if they have no made correct configuration
2. Design Particle class
   1. Create attributes of the particles to allow quarks to be added to the representation, and also to present them to screen
   2. Create methods that check if quark configuration represents and actual particle
3. Design Quark Class
   1. Create class that has attributes that represent the quarks such as name, colour and spin
   2. Link these to buttons in the GUI

# Design

## Class Diagrams from solution:

particleclass.py

|  |
| --- |
| Particle(Entity) |
| quarkpos  name  quarkconfig  partype  charge  spin  strangeness  partIdentify |
| \_\_init\_\_  isSpace(self)  clear(self)  addQuark(self,quark)  sorting\_func(self,quark)  getName(self)  getPartype(self)  getCharge(self)  getSpin(self)  getQuarkpos(self)  getStrangeness(self)  quarkMakeup(self) |

After the creation of the parent class, I created child classes for each of the composite particles (i.e. one for proton, one for neutron). These child classes number at 12, but all share the same attributes, but with different values. As shorthand, I will group these classes into one class called “particles\_type” whose parent class is Particle. Note that there is not a class in my code called “particles\_type” – this is done only for the sake of simplicity.

This is also true for the quark class – I will call these classes “quarks\_type”

|  |
| --- |
| particles\_type(Particle) |
| name  quarkconfig  partype  charge  spin  strangeness |
| None |

quarkclass.py

|  |
| --- |
| Quark(Entity) |
| SIZE X Y name  charge  spin  colour |
| \_\_init\_\_  getName(self)  getCharge(self)  getSpin(self)  getColour(self)  getSize(self)  getThickness(self)  getX(self)  getY(self)  drawCircle(self, screen) |

|  |
| --- |
| quarks \_type(Quark) |
| name  charge  spin  colour |
| None |

particles\_type

**Particle**

quarks \_type

**Quark**

## The design Flowchart of the Simulation:

Checks if user presses “Refresh Button”

Print Error message – “Quark configuration doesn’t exist”

Yes

No

Is quarklist/particle space full?

Not in class

In class

Print data on particle stored in class and in text file on simulation window

Check if list combination of quarklist is in class

Check against particle list in particles\_type class

Print on simulation window & add to the quarklist (excluding “None”)

No input

Checks if user presses Quark/anti-quark button (Inc "None” button)

Get user mouse input

Display simulation window

Start

Pressed

No Input

Sort the values in quarklist/particle – using MergeSort function

Checks if user wants to quit

End

## Methods/Attribute description table:

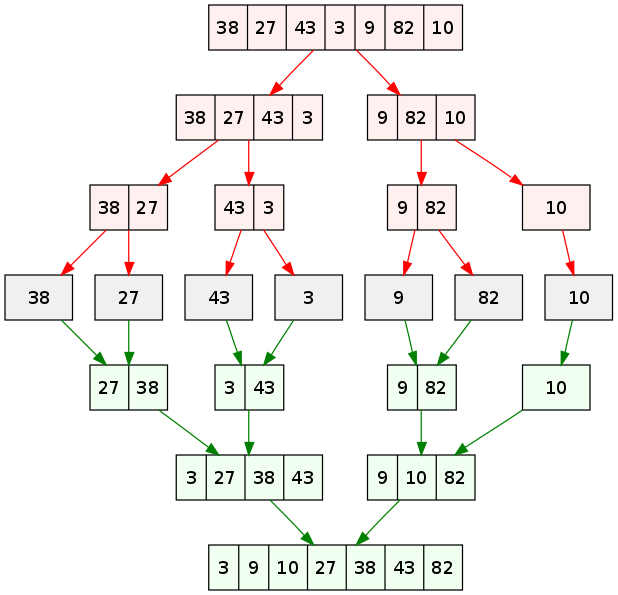
|  |  |
| --- | --- |
| Attribute/method | Description |
| Quarkpos – attribute (Particle class) | Three element list which stores the co-ordinates of all three quark positions 🡪 as a particle can be made up of a maximum of three quarks |
| Name - attribute | Unique identifier used to differentiate between different types of quarks and particles |
| Quarkconfig- attribute (Particle class) | Three element list which stores a list of quarks -> these are stored in the particles\_type subclass; used to identify particles |
| Partype - attribute (Particle class) | Stores a string value which is used to identify the group, which a particle belongs to – i.e. a proton is a baryon. |
| Charge – attribute | A numerical value which represents the electric charge of a particle/quark (stored as a string value as no arithmetic calculations are done with it) |
| Spin - attribute | A numerical value which represents the angular momentum of a particle/quark (stored as a string value as no arithmetic calculations are done with it) |
| Strangeness - attribute (Particle class) | A numerical value which represents the “Strangeness” of a particle/quark (stored as a string value as no arithmetic calculations are done with it) |
| partIdentify - attribute (Particle class) | Three element list which stores the current list of quarks selected by the user (with the attribute starting off as (None,None,None) |
| isSpace(self) - Method (Particle Class) | Checks if there is space in the quarklist, returns either True or False depending on the result |
| clear(self) - Method (Particle Class) | Resets all the attributes to their original values. This method is used in conjunction with the “refresh” button to clear the particle space |
| addQuark(self,quark) - Method (Particle Class) | Adds quark to the quarklist, and appends it. But if it finds that the list if full, it prints out a message notifying the user |
| sorting\_func(self,quark) - Method (Particle Class) | Sorts the elements within the partIdentify list in alphabetical order 🡪 will make use of merge sort algorithm (efficiency) |
| getName(self) – Method | Returns the name of the particle/quark |
| getPartype(self) - Method (Particle Class) | Returns the name of the group which the particle belongs to (e.g. Meson, Baryon) |
| getCharge(self) - Method | Returns the value of the particle’s/quark’s charge |
| getSpin(self) - Method | Returns the value of the particle’s/quark’s spin |
| getQuarkpos(self) - Method (Particle Class) | Returns the co-ordinates of the quark positions (from the quarkpos list) |
| getStrangeness(self) - Method (Particle Class) | Returns the value of the particle’s strangeness |
| quarkMakeup(self) - Method (Particle Class) | Returns the quark configuration of a particle in the form of a list |
| SIZE - attribute (Quark class) | Stores the size of the quark – set at 20 (a constant) |
| X - attribute (Quark class) | Stores X value -`a constant attribute used for testing purposed (set at 100) |
| Y - attribute (Quark class) | Stores Y value -`a constant attribute used for testing purposed (set at 100) |
| colour - attribute (Quark class) | Stores a quark’s colour in the form of a rgb (red,green,blue), three element list |
| getColour(self) - Method (Quark class) | Returns the rgb value of a quark – the colour of a particular quark |
| getSize(self) - Method (Quark class) | Returns the size of a quark – set at 20 |
| getThickness(self) - Method (Quark class) | Thickness and Size are same to allow for a whole circle to be drawn – this method maybe redundant (could just use getSize again) |
| getX(self) - Method (Quark class) | Returns the value of the “constant” x co-ordinate (testing only) |
| getY(self) - Method (Quark class) | Returns the value of the “constant” y co-ordinate (testing only) |
| drawCircle(self, screen) - Method (Quark class) | Draws out a circle – which represents a quark, using values from attributes such as colour,(x,y) and size |

## Sorting Algorithm for quarklist/partIdentify – Merge Sort

Before comparing the current quark configuration to the configurations stored in the Particle subclasses – the quarks will first need to be sorted. I have chosen to implement what is known as a ‘merge sort’. I have chosen this algorithm because it is a greatly efficient algorithm that has a sort time of O(nLog(n)). This is a recursive-sorting algorithm that consists of splitting a list into two, then calling the function again to split the first half, until the final list only has one item. From this point, it will go back and call the function using the second part of each list (starting at the end). This will eventually mean that all the items in the original list are their own lists.

These lists will be compared and joined back together as represented in the diagram below (from

Wikipedia):



Implementation of the algorithm will be done by converting the merge sort pseudo code into python code:

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

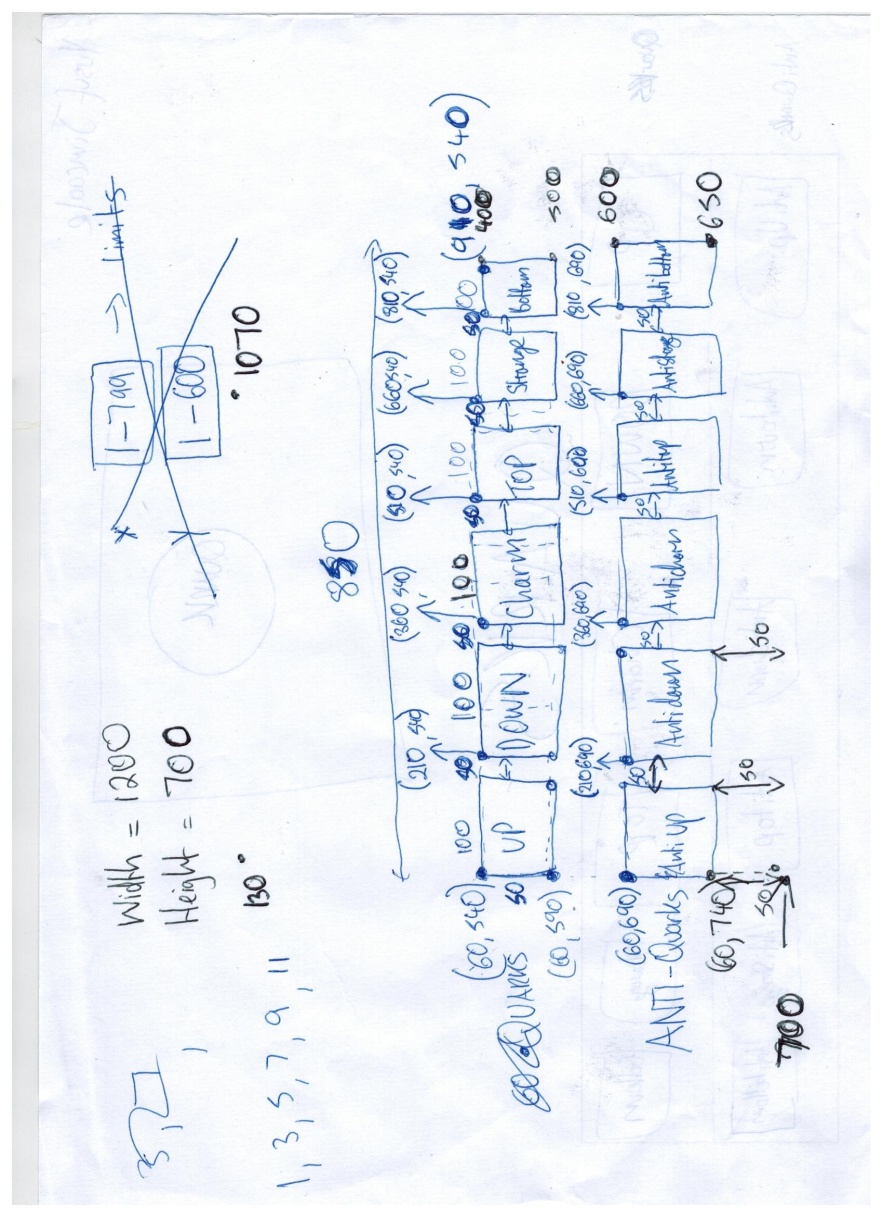
Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:18

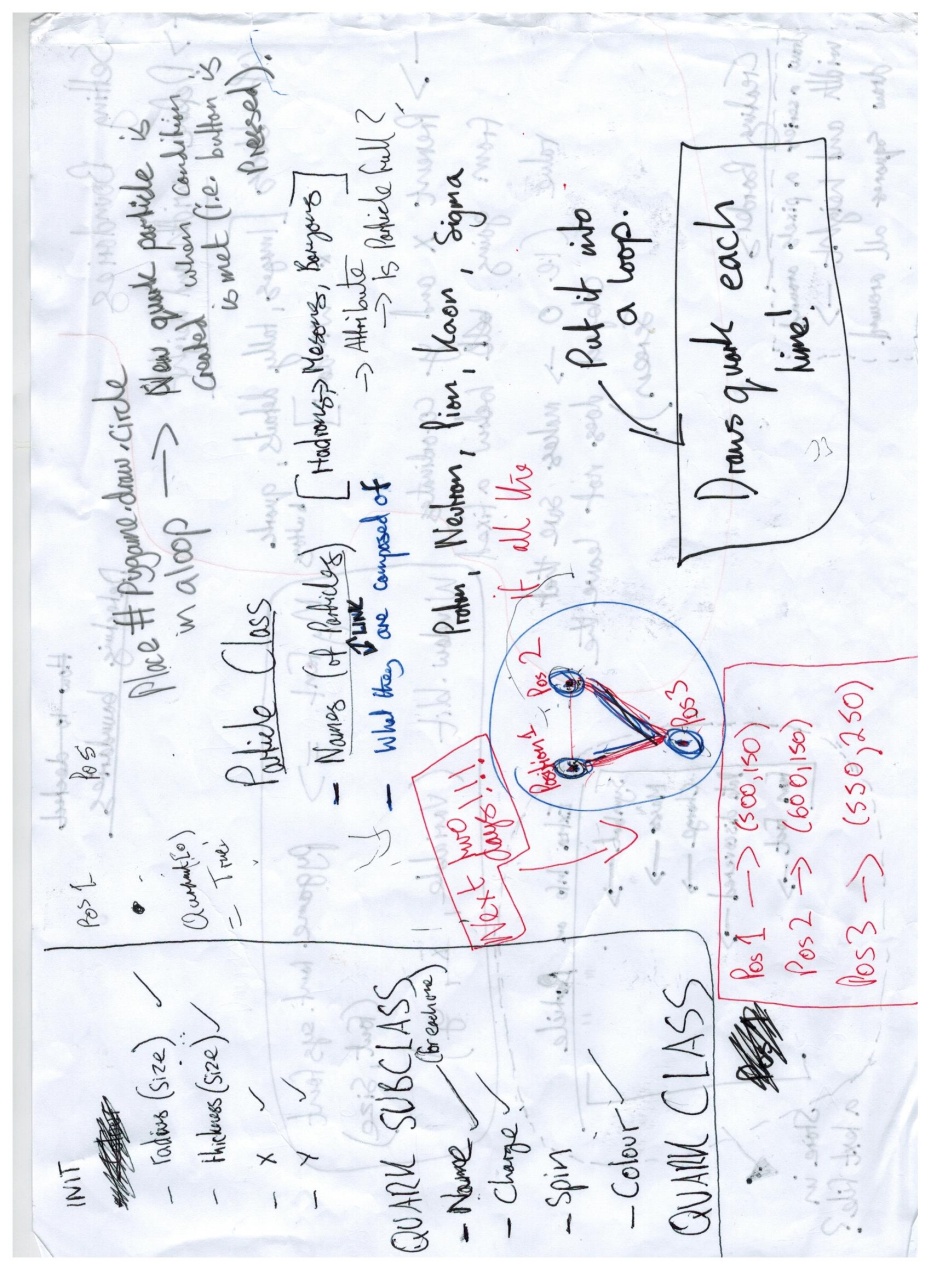
Call merge(arr, l, m, r)



**(https://www.geeksforgeeks.org/merge-sort/)**

## NEA Rough Notes:

Early designs for quark/anti-quark buttons

****

Particle class designs + quark representation

Particle class designs + quark representation

# Image (11).jpg

# C:\Users\Yusuf Jimcaale\Documents\Scanned Documents\Image (14).jpg

# 

Planning out quark subclasses

# 

Measuring and calculating the positioning of borders and quark co-ordinates

## Possible edits to NEA

Edits to make for Analysis/Design:

* Use of references
* What software will be used and WHY
* Make sure it is clear – for third parties to understand
* Merge Sort – discuss in more detail
* Take questionnaire – about how people revise/study for physics
* Discuss about the underlying physics behind quarks and other subatomic particles in detail
* Constraints and limitations
* Investigate existing software (comparisons – in the form of tables) [i.e.Phet lab simulations]
* Discuss algorithms which relate to the behaviour of the particles

Possible Edits for technical solution (TBD – list will be edited throughout the process):

* Make the GUI dynamic – for different resolutions (positions of quark, particle field, buttons etc; should be defined in terms of variables rather than constants)
* Edits bugs
* Introduce a quark tally
* Add extra mouse functionality (dragging particles into the field)
* Add 3-D functionality (needs work)