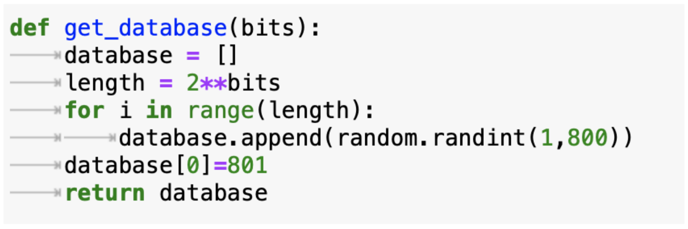
Grover Adaptive Algorithm - Testing

Ana Villarrubia Palacín

March 9, 2023

For the programme *grover_adaptive_cpu.py* we would like do do some testing by, instead of using the actual ESI planets database, generating a random database which will allow us to also test the programme by varying the number of qubits as follows:



```python
def get_database(bits):
	database = []
	length = 2**bits
	for i in range(length):
		database.append(random.randint(1,800))
	database[0]=801
	return database
```

Figure 1: If we are working with a 3-qubit register, we can see that the highgest value in this randomly generated database would of course be $|000\rangle$, with a value of 801.

We are also implementing in this case a new variable called *min_val*, which allows us to look for a range of values rather than a target, and will also do the testing on varying that value. This of course meant deleting all older references to the variable *threshold*.

Since we are looking for a range of values, we expect that the programme is resistent to error, but only would take more time to run when the error probability is larger. I will also add the computation times for each test.

Also sorry because I did not realise that in the first SEVEN figures I wrote dataabse instead of database, if we include this in the report I will change it.

1 Variation of error probability

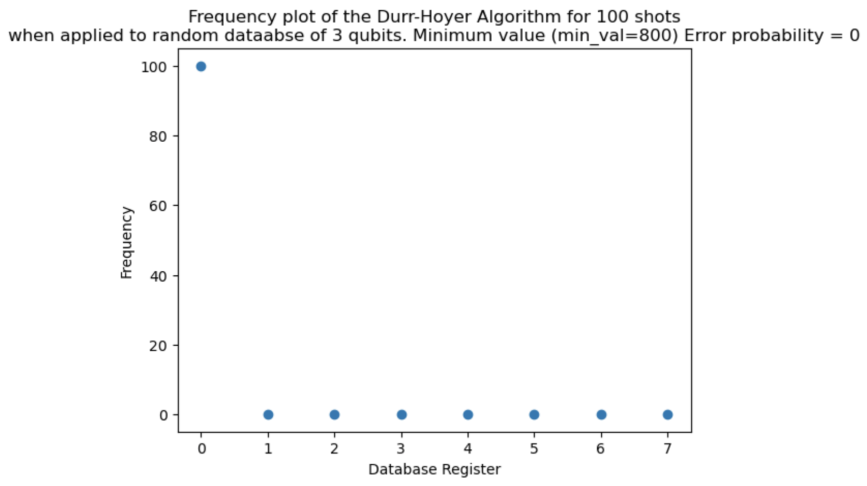


Figure 2: Runtime = 0.39375901222229004s

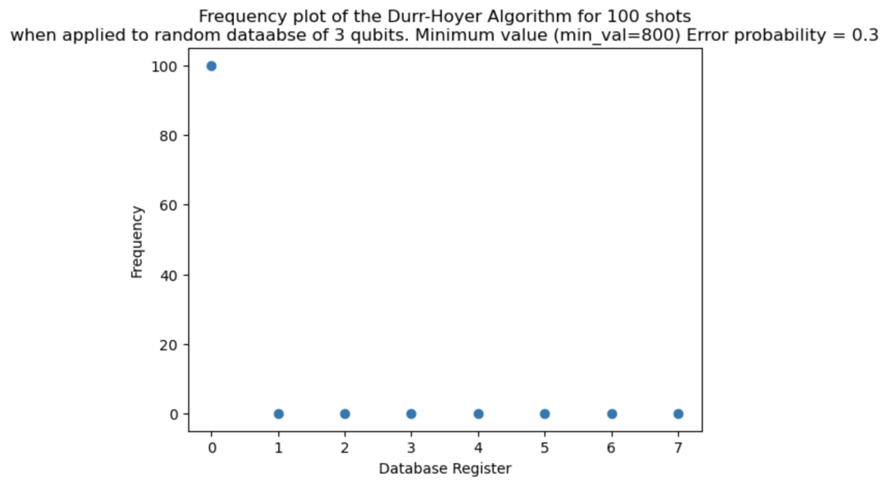


Figure 3: Runtime = 0.6513032913208008s

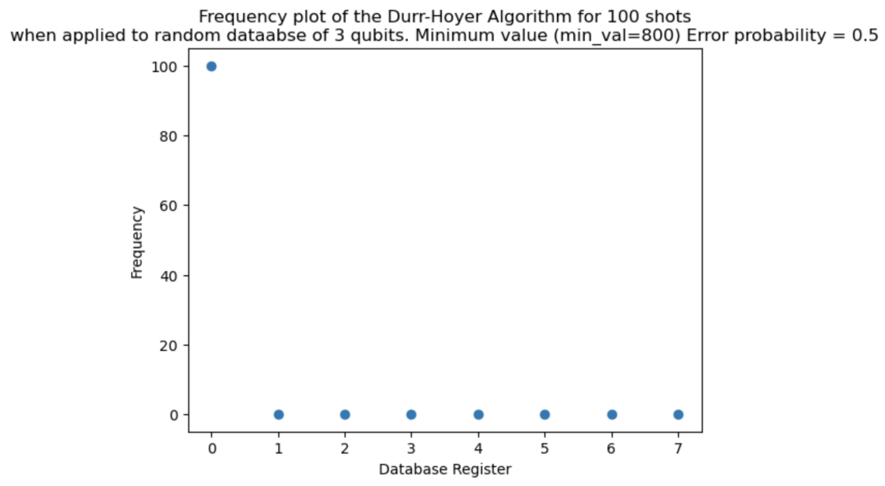


Figure 4: Runtime = 0.8773360252380371s

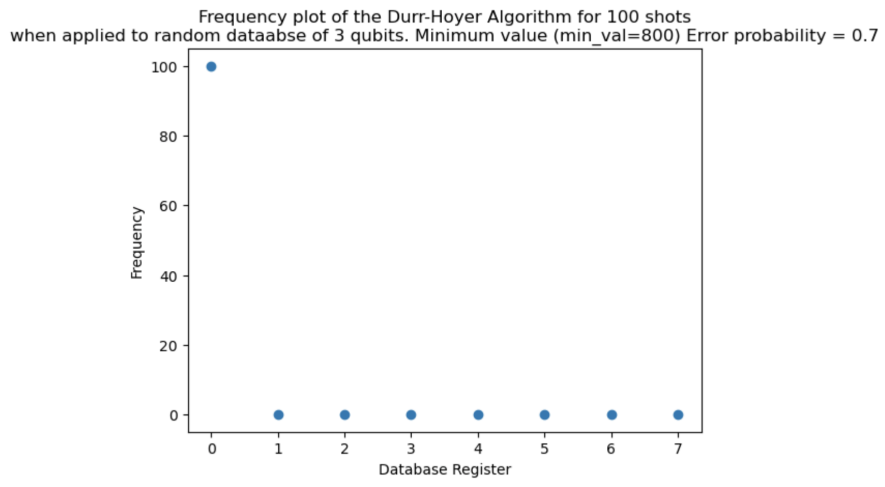


Figure 5: Runtime = 1.0988891124725342s

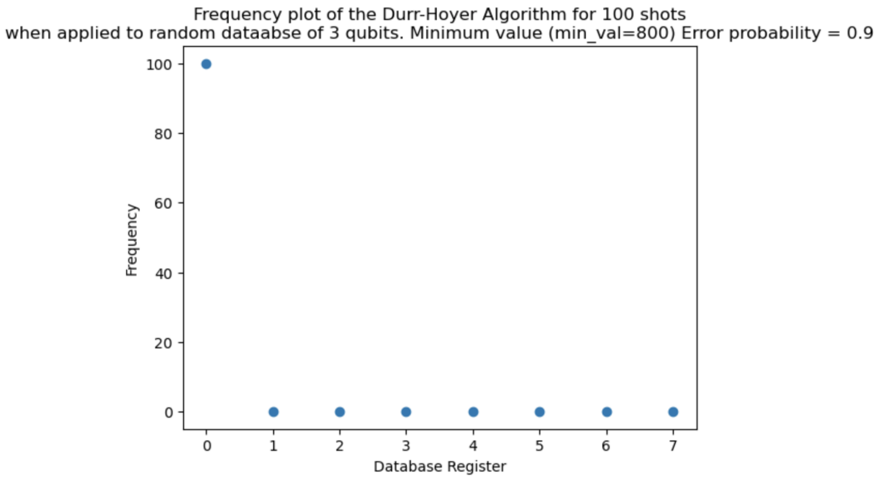


Figure 6: Runtime = 1.5959539413452148s

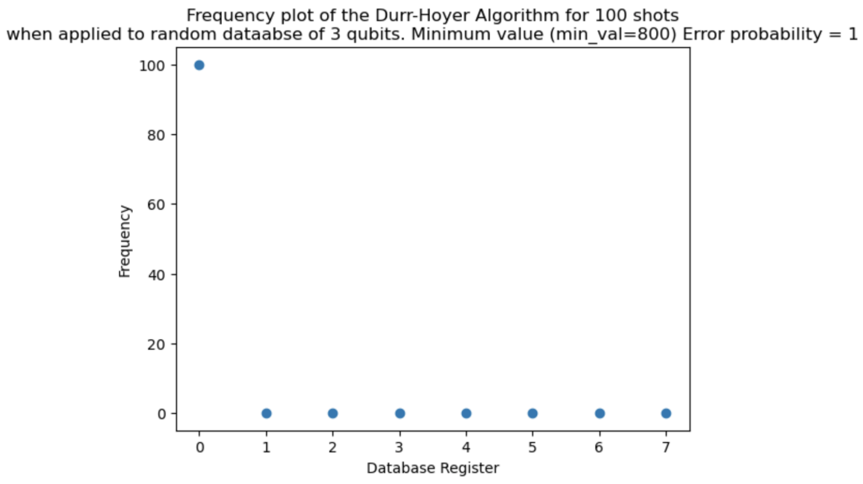


Figure 7: Runtime: 2.3649098873138428s

To not make this report way too long we are not putting all graphs since those ones are enough top demonstrate stability of the programme against errors when looking for a range of values. However, we still need to show the dependence of runtime on the error probability, so I calculated some more values that we can add to the final plot.

When $errorp = 0.1$ we obtained a runtime of 0.42333078384399414 seconds.
When $errorp = 0.2$ we obtained a runtime of 0.5374641418457031 seconds.
When $errorp = 0.4$ we obtained a runtime of 0.7484589862823486 seconds.
When $errorp = 0.6$ we obtained a runtime of 0.9940447807312012 seconds.
When $errorp = 0.8$ we obtained a runtime of 1.3646509647369385 seconds.

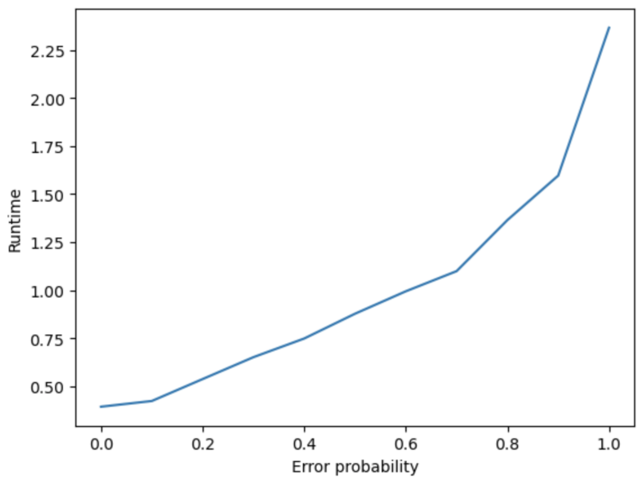
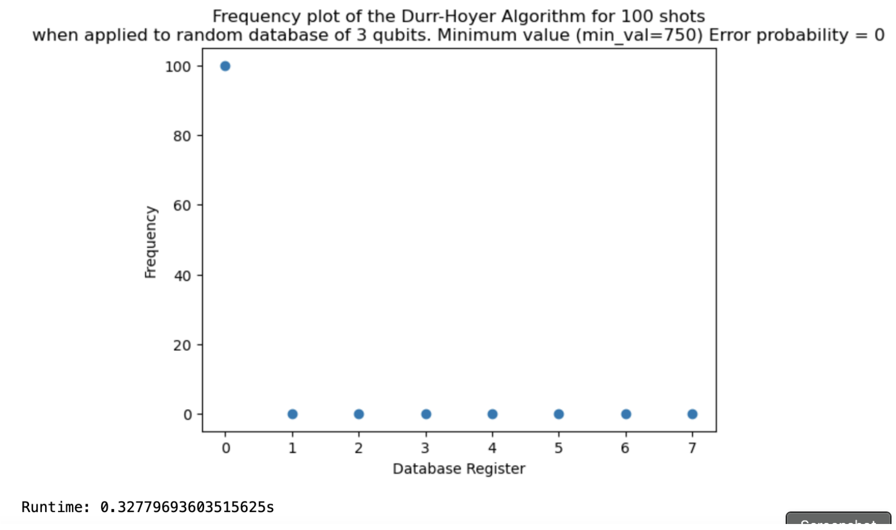


Figure 8: Plot of the runtimes vs the error probabilities obtained.

We can see that varying the error probabilites does indeed vary the runtimes (seems like exponentially, however we would need to do more testing to verify that claim), which is again what was expected.

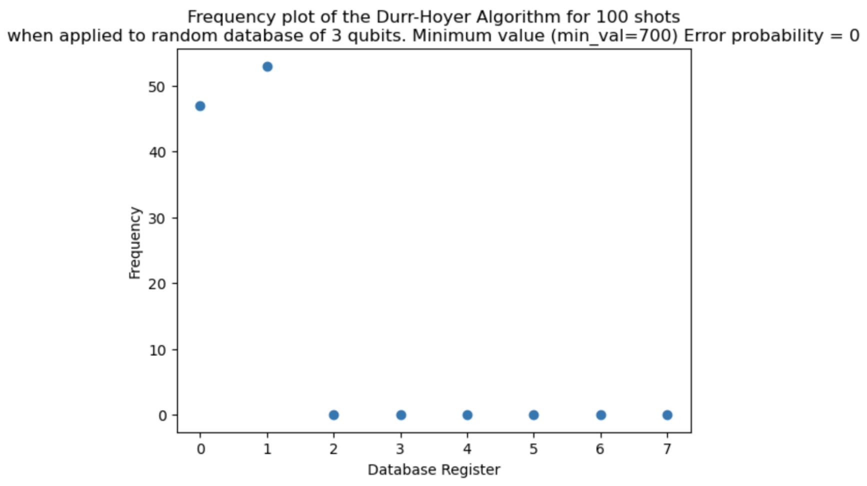
2 Variation of range of values

Using Figure 2 as our first graph for this testing, since it is minimum value = 800 with error probability of also 0.


Frequency plot of the Durr-Hoyer Algorithm for 100 shots
when applied to random database of 3 qubits. Minimum value (min_val=750) Error probability = 0
Frequency
Database Register
100
80
60
40
20
0
0
1
2
3
4
5
6
7

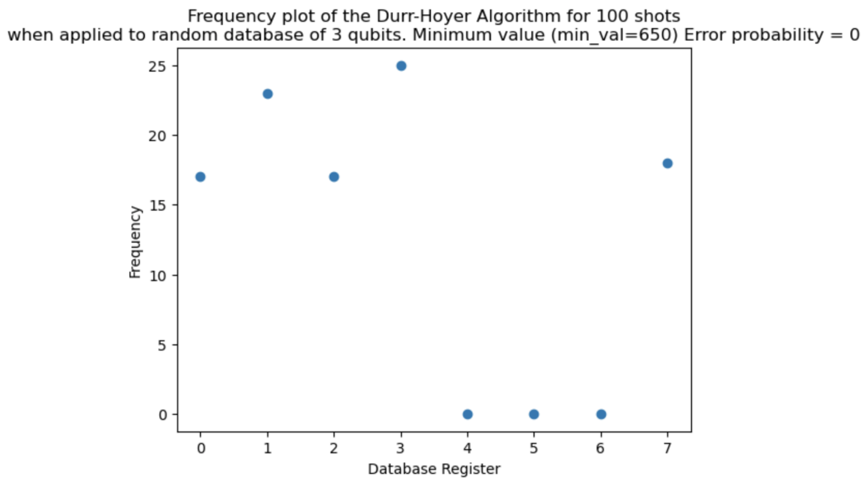
```
Runtime: 0.32779693603515625s
```

Figure 9: minimum value = 750



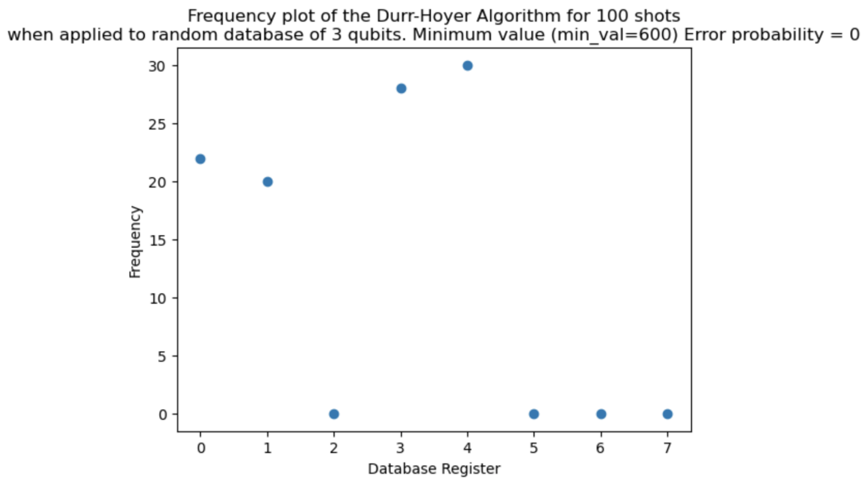
```
Runtime: 0.2613399028778076s
```

Figure 10: minimum value = 700



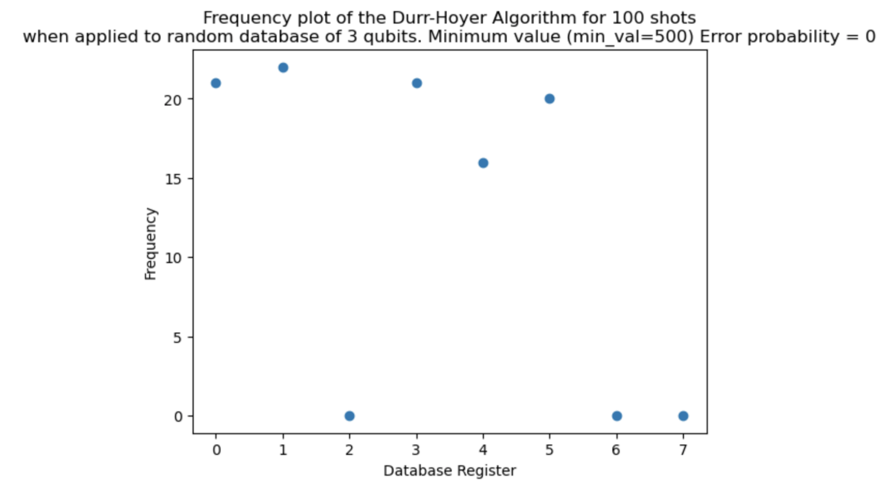
```
Runtime: 0.2530829906463623s
```

Figure 11: minimum value = 650



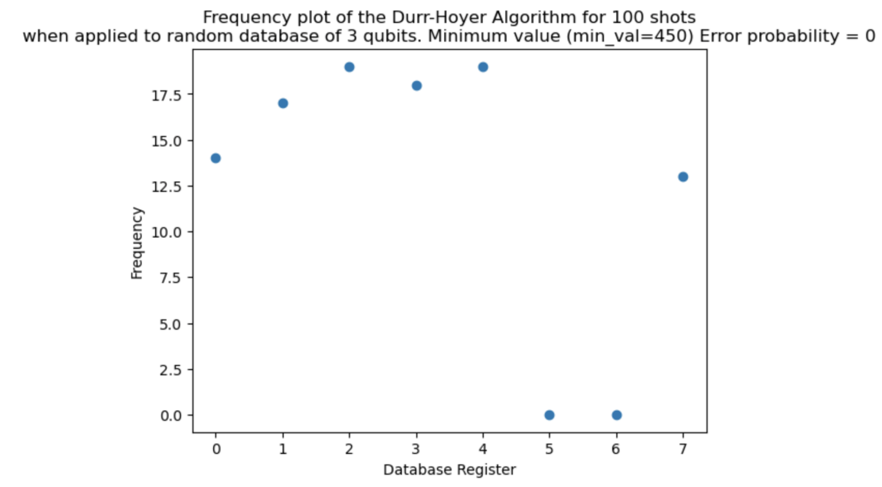
```
Runtime: 0.2469806671142578s
```

Figure 12: minimum value = 600



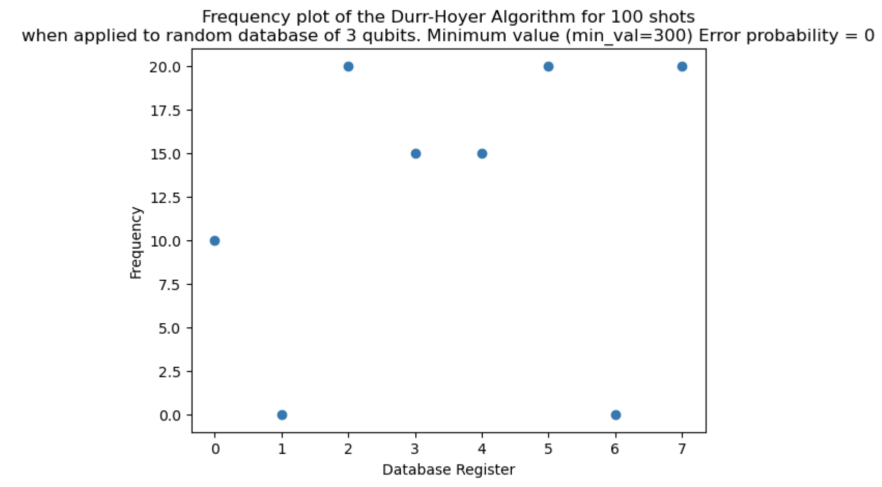
```
Runtime: 0.23736929893493652s
```

Figure 13: minimum value = 500



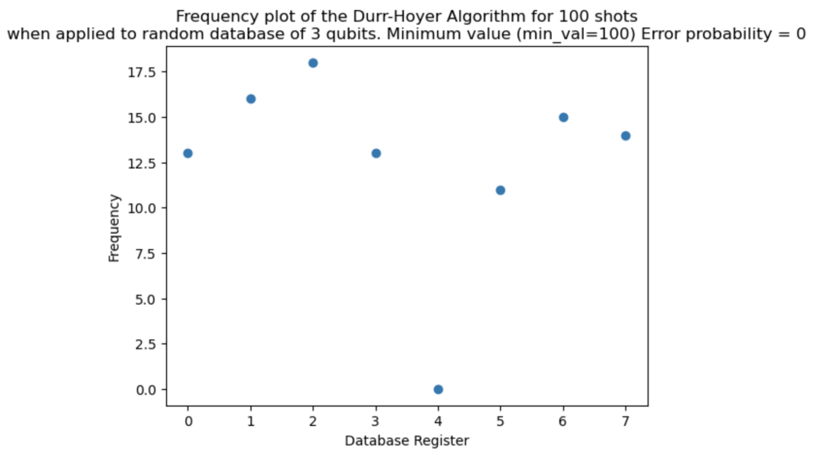
```
Runtime: 0.22565603256225586s
```

Figure 14: minimum value = 450



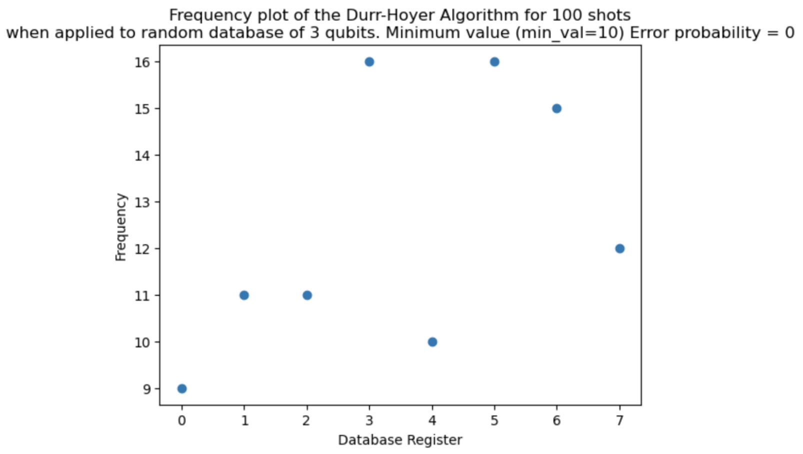
```
Runtime: 0.23198509216308594s
```

Figure 15: minimum value = 300



```
Runtime: 0.2134840488433838s
```

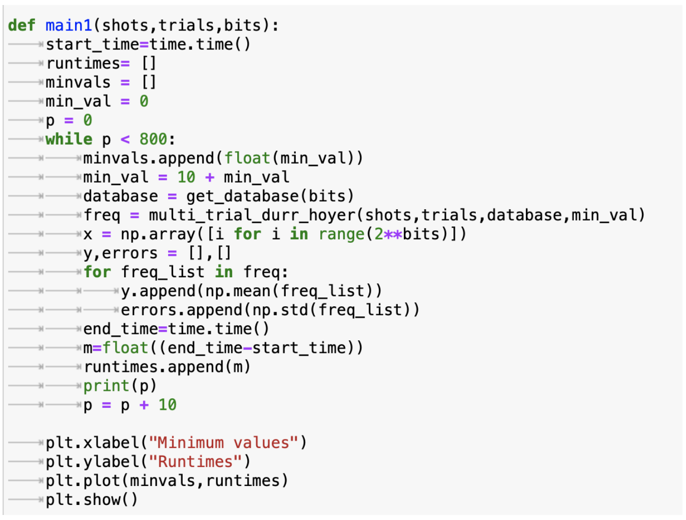
Figure 16: minimum value = 100



```
Runtime: 0.18013906478881836s
```

Figure 17: minimum value = 10

We observe that when the range of values is increased (i.e, *min_val* is decrased) the programme finds more 'possible values' within each shot.
I computed the relationship between the increase of minimum value and the runtimes changing *main_1* to the following:



```python
def main1(shots,trials,bits):
    start_time=time.time()
    runtimes= []
    minvals = []
    min_val = 0
    p = 0
    while p < 800:
        minvals.append(float(min_val))
        min_val = 10 + min_val
        database = get_database(bits)
        freq = multi_trial_durr_hoyer(shots,trials,database,min_val)
        x = np.array([i for i in range(2**bits)])
        y,errors = [],[]
        for freq_list in freq:
            y.append(np.mean(freq_list))
            errors.append(np.std(freq_list))
        end_time=time.time()
        m=float((end_time-start_time))
        runtimes.append(m)
        print(p)
        p = p + 10

    plt.xlabel("Minimum values")
    plt.ylabel("Runtimes")
    plt.plot(minvals,runtimes)
    plt.show()
```

Figure 18: Change of code to compute Figure 19.

Obtaining the following results

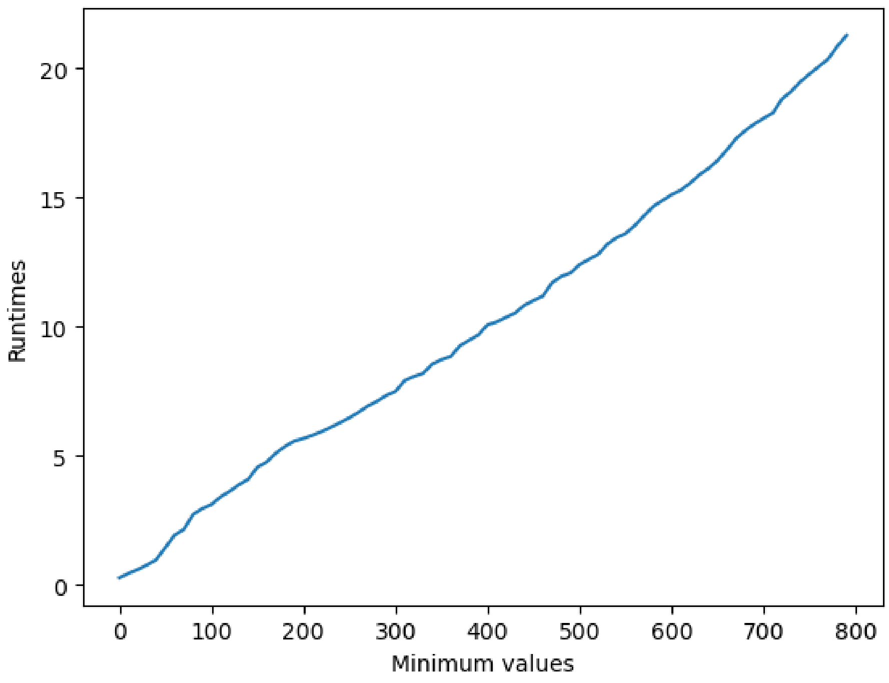


Figure 19: Relationship between minimum value and programme runtime.

As we observe, the runtime increases when the minimum value increases (linearly), this means, that as the range of values gets larger, the runtime decreases, since it takes the programme less time to find target values, which was expected.