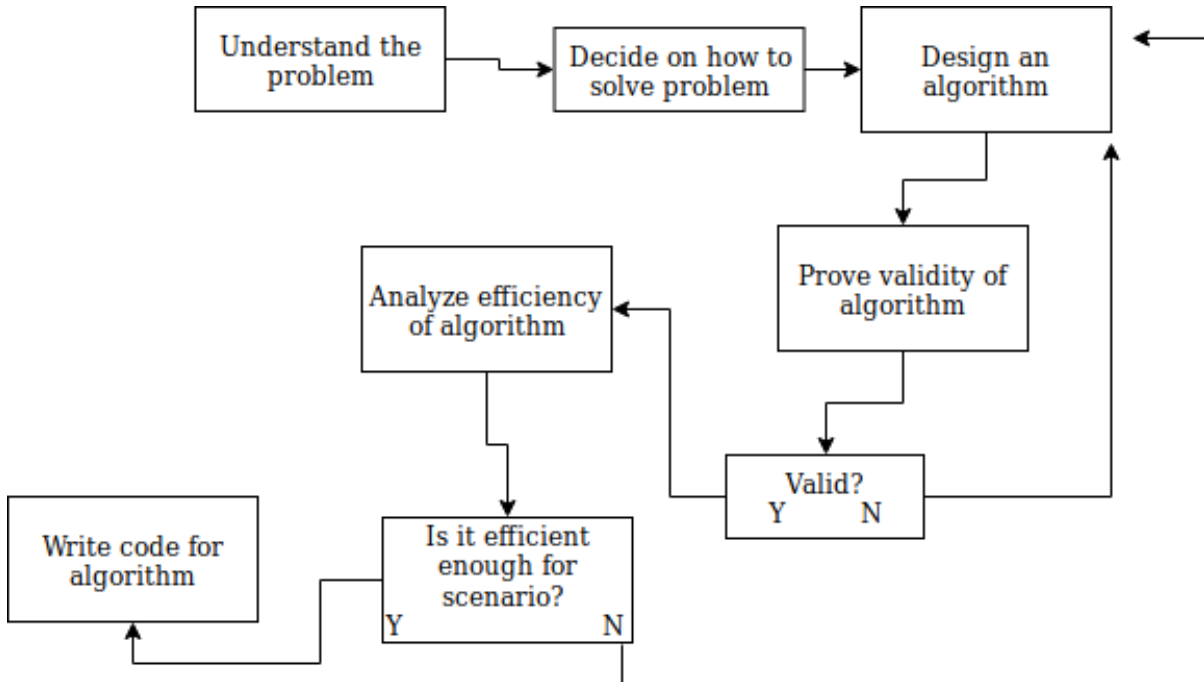1.

Figure 1: Flowchart for design of algorithm

2. An example of a non-basic operation is something that will not count significantly towards the runtime of the program as input size n increases. These include operations such as print statements, variable declarations, etc.

A basic operation is something that contributes significantly to the runtime of the program. This could be as simple as a comparison in a while loop:

```
int n = 0; //non−basic
do{
        cout << "Hey" << endl; //non−basic
        n++; //basic
}while(n<value) //basic
```

In the above code, the lines of code are commented with their basic-ness. Even though the incrementing and the while loop are both considered basic operations, the coefficient drops out when input size n increases, therefore, it does not matter which one is chosen as a basic step.

3. $\log(n), n, n\log(n), n^2, n^3, 2^n, n$

4.1.
a. $\Omega(n)$, in the event that the desired search term is not in the list.
b. $O(1)$, in the event that the desired search term is in index 0.
c. $\Theta(n)$ best represents the average case.

4.2.
a) True
b) True
c) False
d) False

5. The three asymptotic notations:
$O(n)$ (aka big O notation) represents the upper bound of the efficiency of the algorithm.
$\Omega(n)$ (aka big Omega notation) represents the lower bound of the efficiency of the algorithm.
$\Theta(n)$ (aka big theta notation) represents an equal efficiency of the algorithm.

6. $\log(n), n, n\log(n), n^2, n^3, 2^n, n$
Logarithmic, linear, linarithmic, quadratic, cubic, exponential, factorial.

7.
To analyze a non-recursive algorithm:
Decide on a parameter indicating input size.
Find the major significant steps in the program. Steps such as initializing variables are not significant. Steps that are done on each iteration of the algorithm (such as a comparison) are. The goal is to find the step that most significantly effects the performance of the algorithm.
Set up a sum expressing how many times this step is run.
Find a closed form solution for the sum, or calculate its growth rate.

8.
Decide on a parameter indicating input size.
Identify the algorithms basic operation.
Check if the execution count varies on different input sizes.
Set up a recurrence relation with an appropriate base condition for the number of times the basic operation is executed.
Solve the recurrence for a closed form.

9.
MYSTERY
a. This algorithm calculates the sum of squares up to and including some input n.
b. The basic operation is $S \leftarrow S + i \cdot i$
c. The basic operation is computed n times.
d. This algorithm operates in a linear efficiency.

SECRET
a. This algorithm calculates the difference between the maximum and minimum values in an array of numbers.
b. The basic operation is either one of the comparisons.
c. For each iteration (of which there are n-1 of), there are 2 comparisons made. This means that the number of comparisons is $2(n-1)$, which asymptotically rises to show that this operates in a linear efficiency.

ENIGMA
a. This algorithm checks if a 2D matrix is symmetrical about the diagonal axis.
b. The basic operation is $if A[i,j] \neq A[j,i]$.
c. The outer loop operates $n-1$ times, and the inner loop operates between $n-2$ and 1 times.
d. Because this happens inside two nested loops with approximately equal asymptotic sizes, this means that the comparison happens approximately n^2 times. Therefore, this operates in quadratic efficiency.

10.

```
def factorial(n):
        if(n==1):
                return 1
        else:
                return n*factorial(n-1)
```

The basic operation is the multiplication in the bottom line.
Let $M(n)$ represent the number of multiplications done.
$M(n) = M(n-1) + 1$
The base case is when no multiplications are done with $n = 0$. Therefore, $M(0) = 0$.
Use a backward substitution:

$$M(n) = M(n-1) + 1$$

$$M(n) = M(n-2) + 1 + 1$$

$$M(n) = M(n-3) + 1 + 1 + 1$$

A pattern is apparent:

$$M(n) = M(0) + n$$

As the base case has $M(0) = 0$, this simplifies to:

$$\boxed{M(n) \in \Theta(n)}$$

11.

ALGORITHM: FIND IN PHONEBOOK

```
//Assume phonebook has names listed in alphabetic order with no duplicates
//Assume n names are in book
//Assume name to be found is called "guess"

//Declare pointers to the following indexed entries
left = 0
right = n−1
middle = ceil(n/2)


*****

if guess == middle:
        return guess
fi

if guess < middle:
        right = middle
        middle = ceil(right/2)
        GOTO *****
fi


if guess > middle:
        left = middle
        middle = ceil(left/2)
        GOTO *****
fi
```

The worst case is where the node to be found is at the beginning or the end of the list.
Because the algorithm halves the search space each time it runs, the algorithm runs in $O(\log n)$.

12.

ALGORITHM: CHECK IF SORTED

```
for item in list(0,i−1):
        if item[i] >  item[i+1] then
                return false
        fi
return true
```

The worst case scenario in this instance is having the last 2 items be out of order. This means that in a list of length n, the program must do $n - 1$ comparisons.

Let $C(n)$ represent the number of times the basic operation is executed.

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

$$n - 1 \in \Theta(n)$$

The time efficiency is $\Theta(n)$

13.

```
ALGORITHM INSERTION SORT
i = 1
while i < size:
        j=1
        while j>0 and A[j-1]>A[j]
                swap A[j] and A[j-1]
                j--
        elihw
        i++
elihw
```
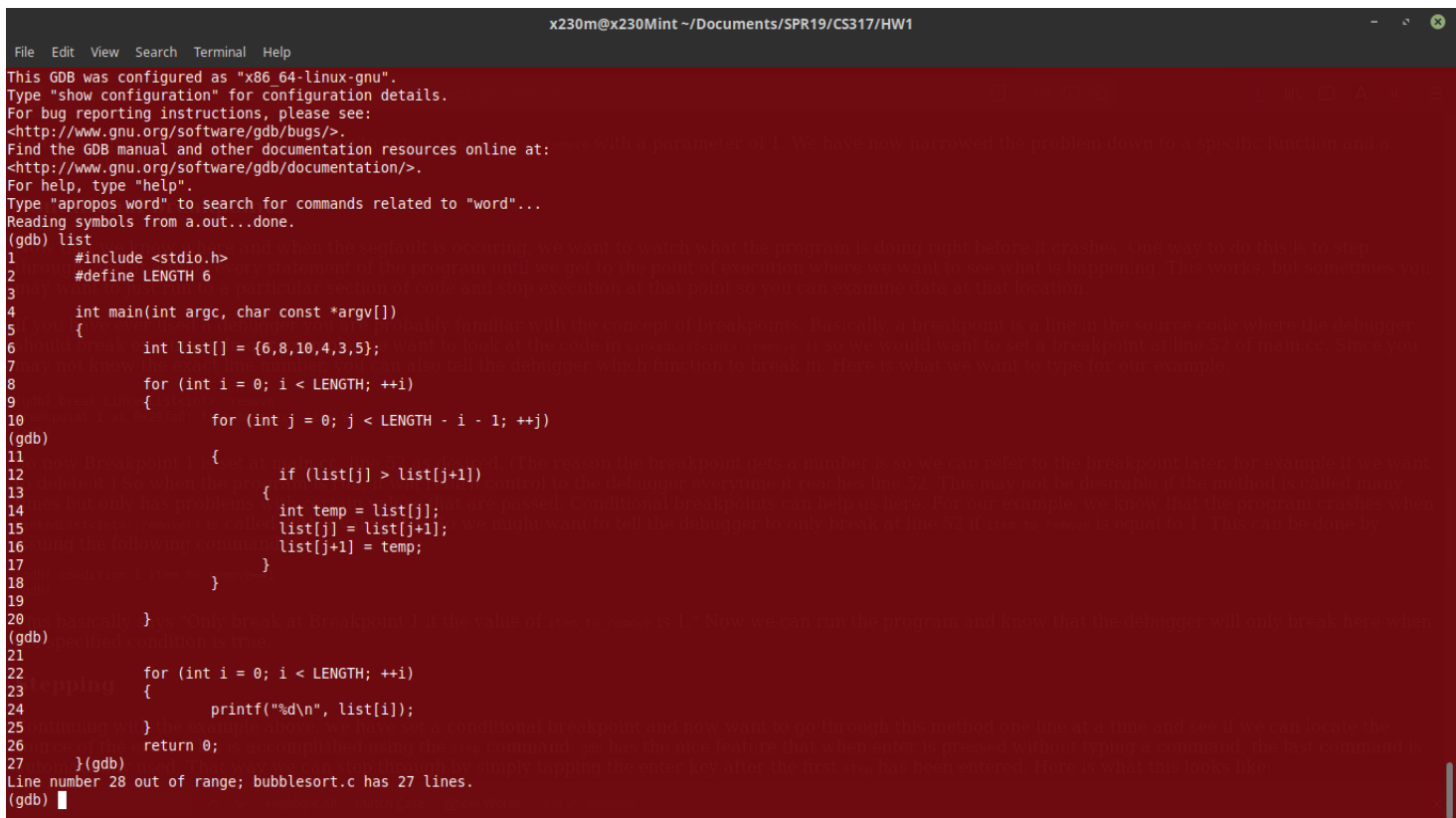
Insertion sort runs in $O(n^2)$ efficiency in worst case.
Its average case is also $O(n^2)$ efficiency.
The best case scenario is when the list is already sorted, which uses $O(n)$ comparisons and $O(1)$ swaps.
Information obtained from wikipedia article on insertion sort.

---

14. I do not use an IDE to develop my code. However, Linux has a command line tool called gdb (Gnu DeBugger) that I can use in the same way.

Figure 2: Screenshot of debugger tool showing code.

```
7
8              for (int i = 0; i < LENGTH; ++i)
9              {
10                   for (int j = 0; j < LENGTH - i - 1; ++j)
(gdb)
11                   {
12                        if (list[j] > list[j+1])
13                        {
14                             int temp = list[j];
15                             list[j] = list[j+1];
16                             list[j+1] = temp;
17                        }
18                   }
19
20             }
(gdb)
21
22             for (int i = 0; i < LENGTH; ++i)
23             {
24                  printf("%d\n", list[i]);
25             }
26             return 0;
27      }(gdb)
Line number 28 out of range; bubblesort.c has 27 lines.
(gdb)
Line number 28 out of range; bubblesort.c has 27 lines.
(gdb)
Line number 28 out of range; bubblesort.c has 27 lines.
(gdb)
Line number 28 out of range; bubblesort.c has 27 lines.
(gdb) break 12
Breakpoint 1 at 0x704: file bubblesort.c, line 12.
(gdb) condition 1 i==2
(gdb) run
Starting program: /home/x230m/Documents/SPR19/CS317/HW1/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at bubblesort.c:12
12                        if (list[j] > list[j+1])
(gdb) print list
$1 = {6, 4, 3, 5, 8, 10}
(gdb)
```

Figure 3: A breakpoint has been added, and the list has been printed in its current state.

```
(gdb) list
7
8              for (int i = 0; i < LENGTH; ++i)
9              {
10                   for (int j = 0; j < LENGTH - i - 1; ++j)
11                   {
12                        if (list[j] > list[j+1])
13                        {
14                             int temp = list[j];
15                             list[j] = list[j+1];
16                             list[j+1] = temp;
(gdb) break 12
Breakpoint 3 at 0x555555554704: file bubblesort.c, line 12.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
3       breakpoint     keep y   0x0000555555554704 in main at bubblesort.c:12
(gdb) run
Starting program: /home/x230m/Documents/SPR19/CS317/HW1/a.out

Breakpoint 3, main (argc=1, argv=0x7fffffffdfb8) at bubblesort.c:12
12                        if (list[j] > list[j+1])
(gdb) print(list[j]>list[j+1])
$2 = 0
(gdb)
```

Figure 4: Showing that a breakpoint has been added at the given point in the code, and the comparison value is false.
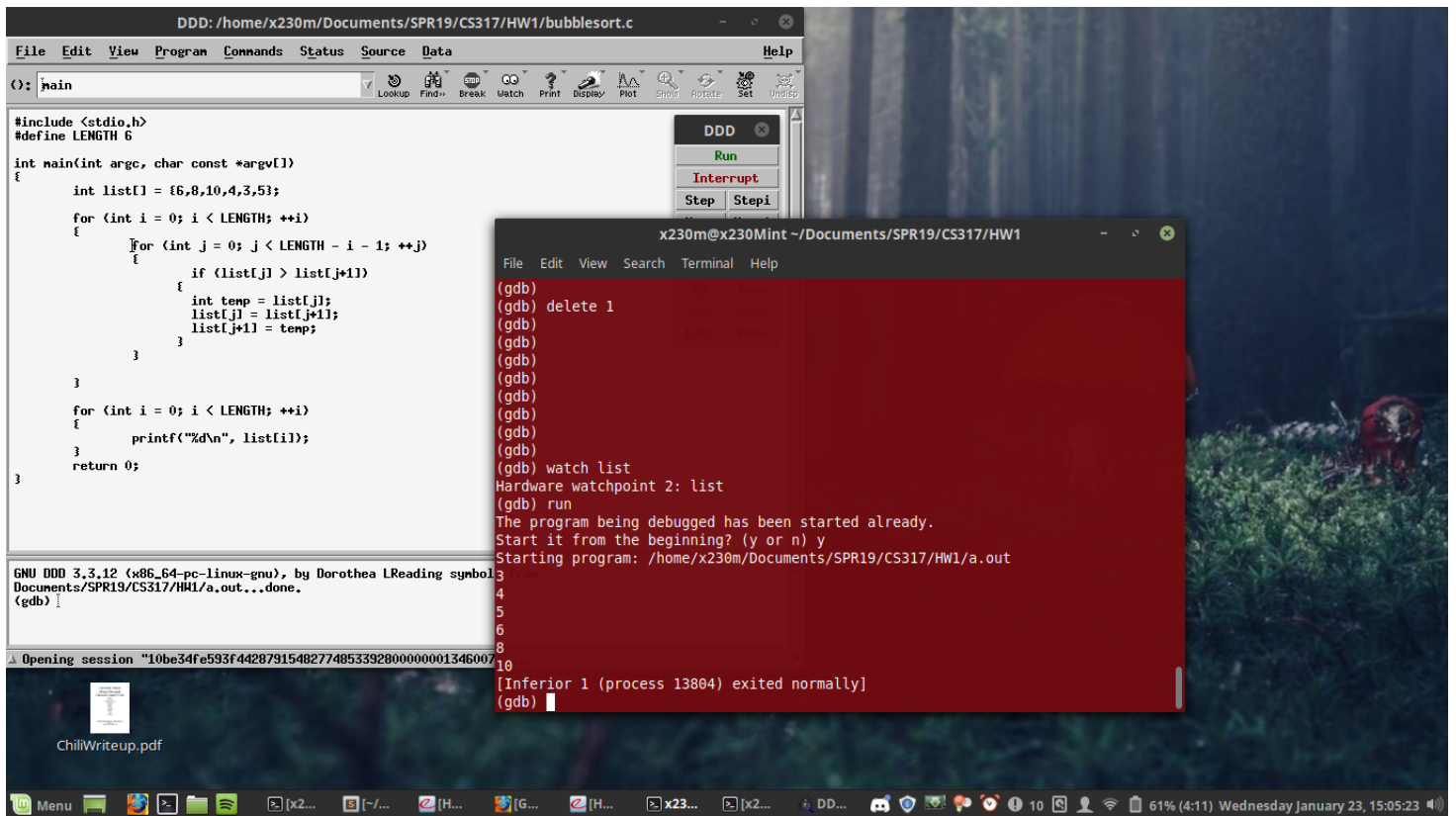
Figure 5: The GUI for this program showing the code and the watchpoints.

15.

The code is attached in file bruteforce.py.

Length	Run Time (seconds)
1	3.7E-5
2	1.44E-3
3	5.38E-2
4	1.83E0
5	5.91E1
6	1.53E3
7	3.98E4

Calculating the 7 character combinations took approximately 11 hours to run.

I would have run it longer, however, I calculated that at the current rate of growth, calculating every possible 35 character password would have taken past the heat death of the universe to calculate.

As I assume that an extension of 10^{35} years would not be acceptable, I decided to simply truncate the table.