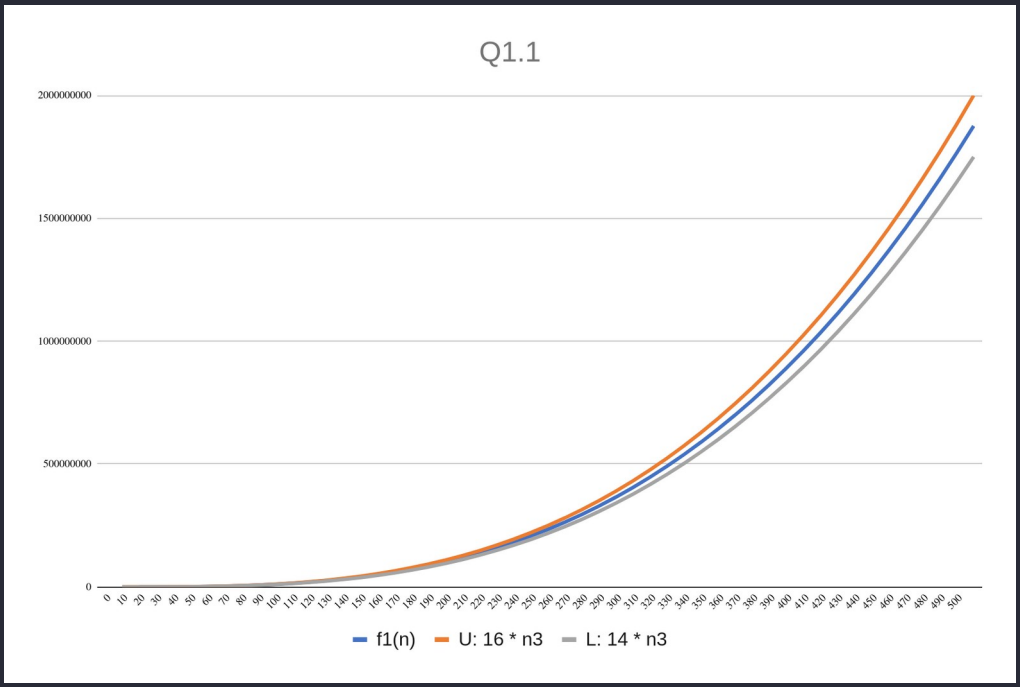
Program Analysis : Assignment One

Q1.

$f_1(n)$:

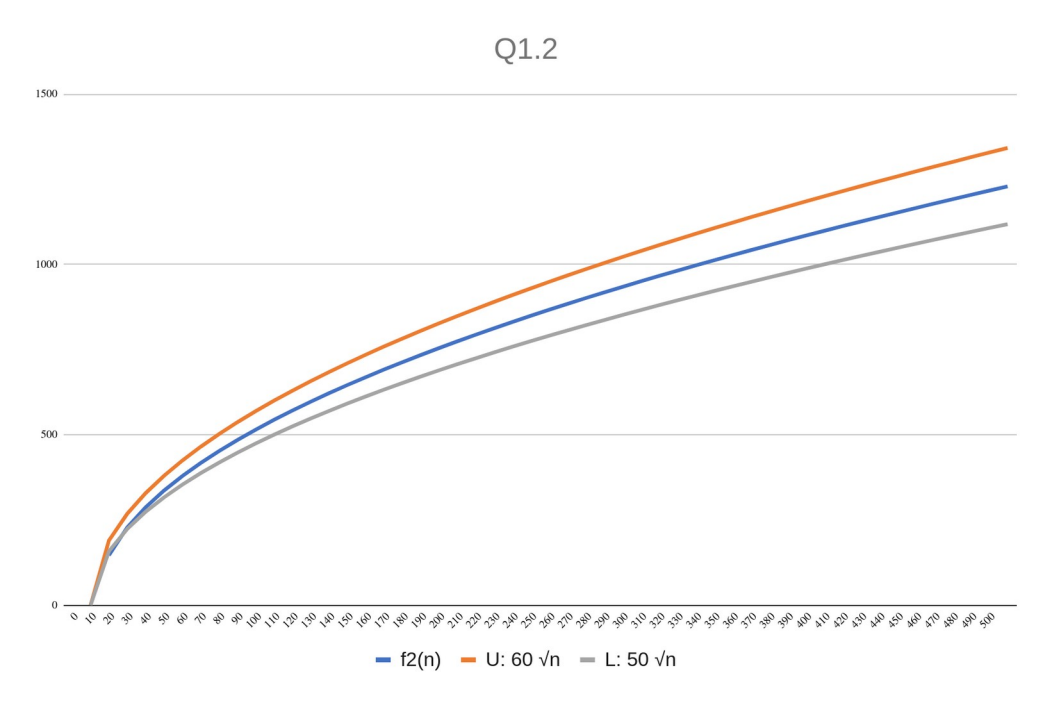


Graph Clarification: n3 means n^3

Bounds: $O(n^3)$ $\Omega(n^3)$ → $\theta(n^3)$

Reason: To find an upper bound, I first took the largest significant term from the equation which is n^3, from there I tested if adding a multiplication bigger than the 15 with the n^3 in the equation would surpass the function and stay above continuously and it did, as this worked I used it for my upper bound, $16n^3$. From there I chose $14n^3$ to see if this worked as a lower bound, I plotted it and never rose above the function, this was my lower bound. They both shared n^3 in common so I could create a tight bound with n^3.

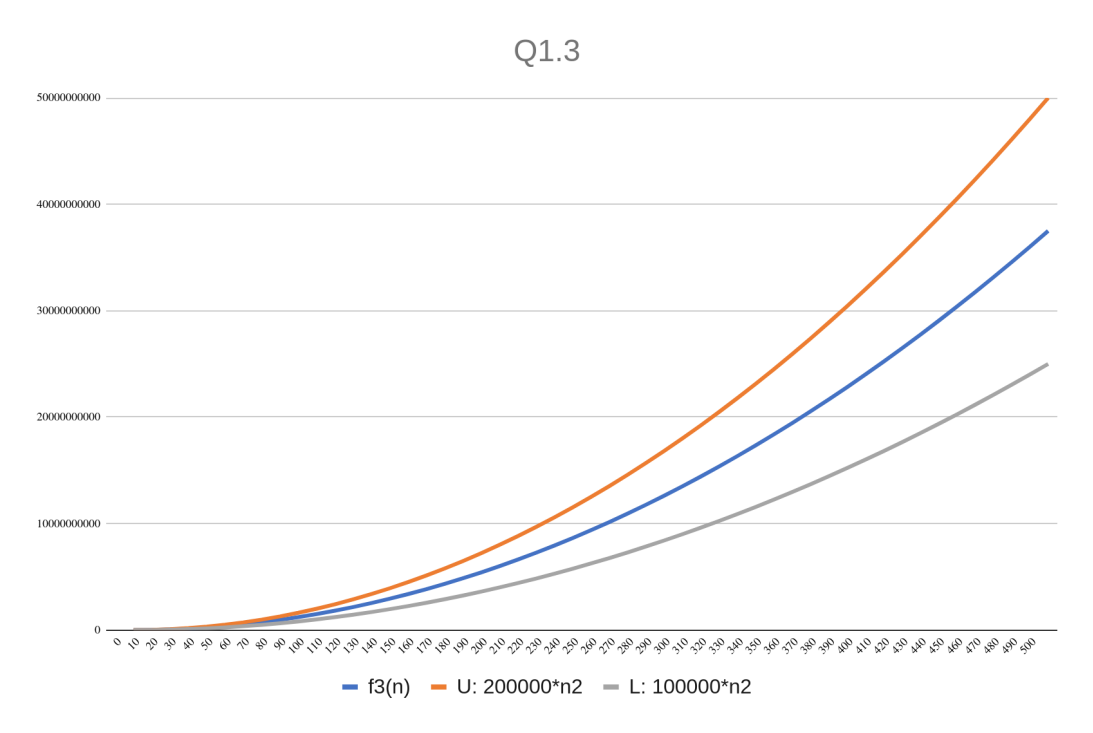
$f_2(n)$:



Bounds: O(√n) Ω(√n) → θ(√n)

Reason: I took the largest significant value, √n, and put a larger multiplicator than 55, which was 60 and this was chosen as my upper bound, my lower bound was chosen through the same way but finding a lower multiplier which was 50. The upper bound never fell below the function, and the lower bound never rose above the function. They shared √n so a tight bound of √n could be applied.

$f_3(n)$:

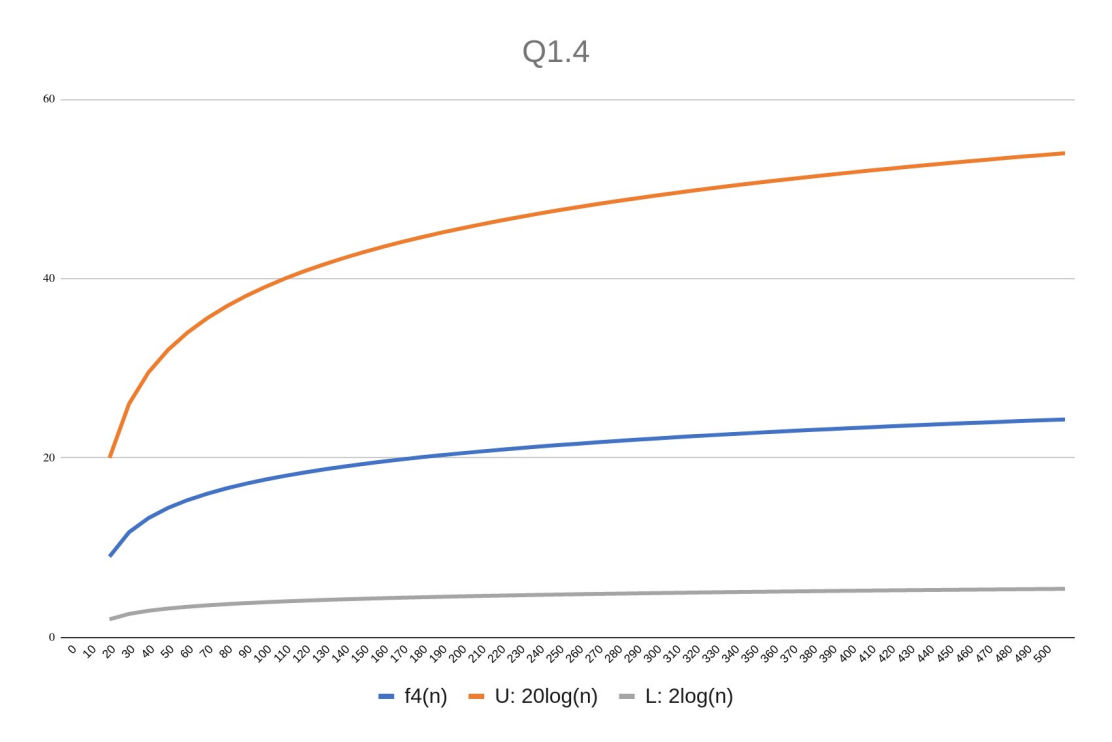


Graph Clarification: n2 means n^2

Bounds: $O(n^2)$ $\Omega(n^2)$ → $\theta(n^2)$

Reason: The most significant value is n^2, I chose this value as the base and made the multiplier higher than the functions for the upper bound, for the lower I took away from the multiplier. The result was where the upper bound doesn't go below the function and the lower bound stays below the function. As they both share n^2 a tight bound can be created which would also be the same value.

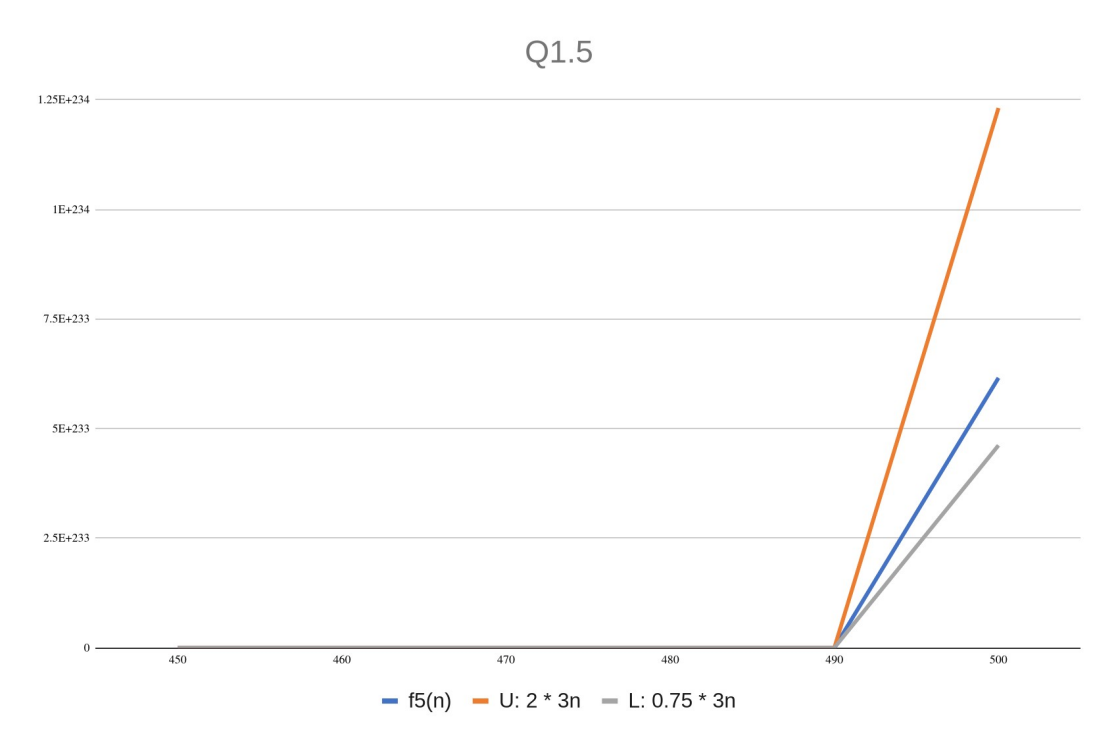
$f_4(n)$:



Bounds: O(n log n) Ω(n log n) → θ(n log n)

Reason: The most significant value is $\log(n^3)$, so I changed the n^3 to just n and changed the multiplier to be 20 for the upper bound which made it above the function. For the other, I chose the multiplier 2, they both share the complexity of log(n) so a tight bound can be created with the same value.

$f_5(n)$:

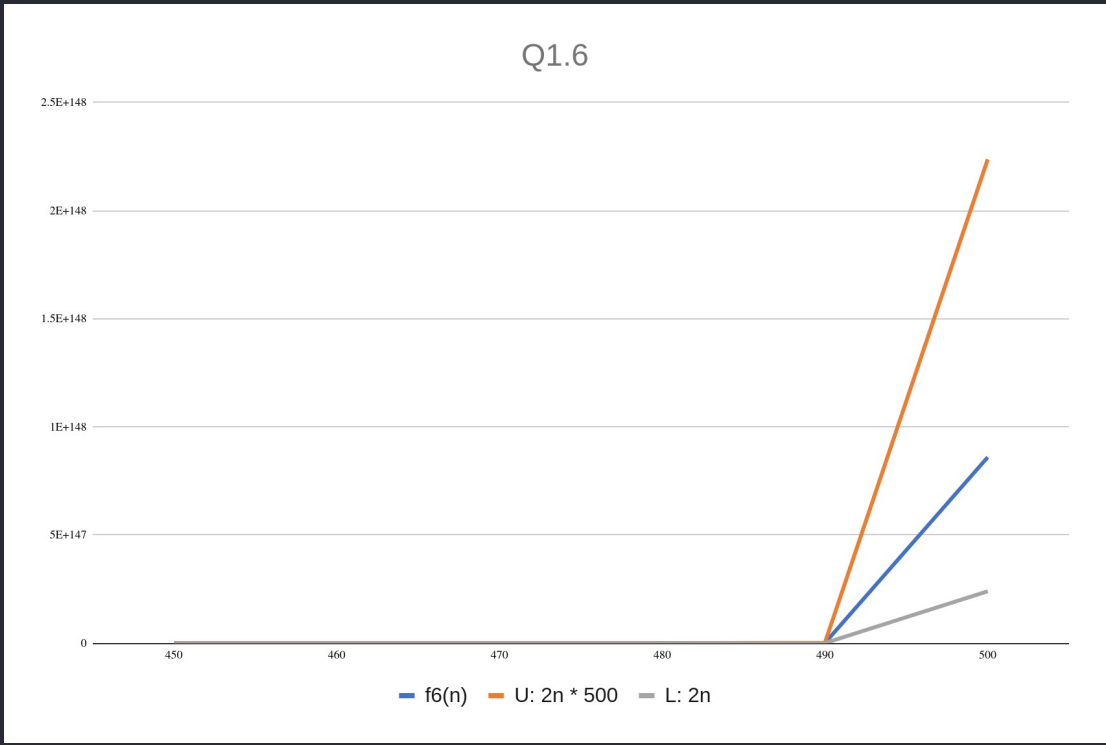


Graph Clarification: 3n means 3^n

Bounds: $O(3^n)$ $\Omega(3^n) \rightarrow \theta(3^n)$

Reason: The most significant value is 3^n so I decided to multiply it by 2 to see if it doesn't eventually get over taken and after 500 steps it has not and the trajectory does not show the possibility of so, the lower bound used the most significant value and multiplied it by 0.75 and has not overtaken the function. The upper and lower bound shares the complexity 3^n which means a tight bound can be created with the same value.

$f_6(n)$:



Graph Clarification: 2n means 2^n

Bounds: $O(2^n)$ $\Omega(2^n) \rightarrow \theta(2^n)$

Reason: Largest significant value is 2^n the upper bound is the most significant value multiplied by 500 which was did not dip below the function, and the lower bound, set as 2^n never went above the function either, i selected this by just removing the multiplier to get a line which had the same significant value as the others. Both upper and lower bound shared the same complexity which means tight bound can be made with the same value.

The reason why in all of these functions we disregard the multiplier or other factors which are not the most significant value is because over time the other terms become irrelevant when growing.

Order :

1. $f_4(n)$
2. $f_2(n)$
3. $f_3(n)$
4. $f_1(n)$
5. $f_6(n)$
6. $f_5(n)$

Q2.

a)

O(n) Ω(1)

Upper: The upper bound is O(n) because there is one for loop present, which runs to *n*, which has the complexity of linear time.

Lower: Ω(1) if the loop is only run once due to *n* being 1 causes constant time due to every line only being run once.

Tight bound is not possible.

The algorithm gets the sum of all numbers in *a* which are bigger than *b*, which is returned.

b)

O(n) Ω(1)

Upper: O(n), this is the case due if both *n* and *m* is greater than 1 in the for loop as the min will select either of the two values to loop to which has to be one or the other, and if both are above the value it will cause the complexity to be linear

Lower: Ω(1), as if one of *n* or *m* is 1 and the other is above it, the min function will select the value of 1 causing the complexity to become constant due to the loop only being processed once like the rest of the code outside of the loop.

Tight bound not possible.

The algorithm adds all of the values together, up to the length of the smallest array, the items in array *a* which are bigger than an item in array *b* and returns it.

c)

$O(n^3)$ Ω(1)

Upper: $O(n^3)$, if *n* if greater than 1, the loop would have to be processed more than once, and as there are three levels of nested loops the resulting complexity would be to the power of three, hence n^3.

Lower: Ω(1), if *n* is 1, Each loop will only be processed once due to the bound of *n*, causing it to act as a constant time complexity.

Tight bound not possible.

The algorithm gets the product of every combination of array item to the power of two, excluding multiplying against any other item in its own array, and then sums up each of the products calculated in *x* to be returned.

d)

$O(n^2)$ $\Omega(1)$

Upper: If n is bigger than 1 and the first two numbers are greater than 0 in array *a* at the minimum, the worst possible is every number being lower than 0 but this won't affect the complexity in this case,

Lower: If the first item in array *a* is less than 0, or if the length of array *a* is one to make *n* one, and the length of array *b* is one to cause *m* to be one as well will cause everything to only be carried out once causing a constant time complexity.

Tight bound not possible.

This algorithm, for every item in array a up to the first item which is less than 0, it then sums up each possible multiplication of the current item in *a* at position *i* in the array with each item in *b* at position after, and including, *i*. This sum is then returned.

e)

Proof by contradiction, that a sum of two odd numbers is odd:

Assume x and y are odd numbers , $x, y \in \mathbb{N}$

And that $x + y = 2k + 1$

For some $k, n \in \mathbb{N}$

As y is odd, it can be written as $y = 2n + 1$

$x + 2n + 1 = 2k + 1$

$x = 2n - 2k$

$x = 2(n - k)$

This contradicts the statement, as x is a multiple of two makes x even due to the multiplication being applied on a natural number, n - k, making it even. Therefore, the sum of two odd numbers is always even.

Q3.

a)

$O(2^n)$, $\Omega(1)$

Upper: There are two possible choices per bit, either 1 or 0, for the worst case we will assume that the algorithm will have to try every combination and the last is the correct, this means that for each bit you need to try each combination creating the 2^n complexity.

Lower: Unfortunately there is always the chance that someone can be lucky and guess the key the first time, making it have a linear time complexity.

b)

We should not be concerned as there is **always** the chance that someone could just be lucky enough that they get the key first time instantly, there is no way of mitigating this as there is always this chance regardless of the algorithm.

c)

$$\frac{((60 \cdot 60) \cdot 24) \cdot 30}{30} = 86400$$

$$\frac{86400}{2^n} < 0.01$$

$$86400 < 0.01 \cdot 2^n$$

$$8640000 > 2^n$$

$$\log_2(8640000) > n$$

$$23.04 > n$$

Therefore, as you can only have a whole as a bit, the minimum amount of bits which is no more than 1% chance that the message would be decoded in 30 days is 24.

d)

```python
# Python3 code
thirty_days_in_seconds = (((60 * 60) * 24 ) * 30 )

a = 100
n = 0

a_total = 0
while a_total < thirty_days_in_seconds:
   t = ( 0.01 * a ) * ( n ** 2 )
   a_total +=  t * ( 2 ** n )
   n += 1

n -= 1

b_total = 0
for i in range(n):
   b_total += 2 ** i

print(b_total - 1)
```

The returned value from the code is: 16382

$$\frac{16382}{2^n} < 0.005$$

$$16382 < 0.005 \cdot 2^n$$

$$3276400 > 2^n$$

$$\log_2 (3276400) > n$$

$$21.63 > n$$

Therefore, as bits can only be a whole integer, the minimum number of bits is 22.

Q4.

a)

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Iteration	**0**	∞	∞	∞	∞	∞	∞	∞
1	0 (i)	**5**	∞	∞	∞	∞	**2**	∞
2	0 (i)	5	∞	**3**	∞	∞	2 (i)	**6**
3	0 (i)	5	∞	3 (i)	**6**	∞	2(i)	6
4	0 (i)	5 (i)	**7**	3 (i)	6	∞	2(i)	6
5	0 (i)	5 (i)	7	3 (i)	6 (i)	**10**	2 (i)	**4**
6	0 (i)	5 (i)	7	3 (i)	6 (i)	**5**	2 (i)	4(i)
7	0 (i)	5 (i)	7	3 (i)	6 (i)	5 (i)	2 (i)	4 (i)
8	0 (i)	5 (i)	7 (i)	3 (i)	6 (i)	5 (i)	2 (i)	4 (i)

(i) = invariant, int = current

I think this answer is correct.

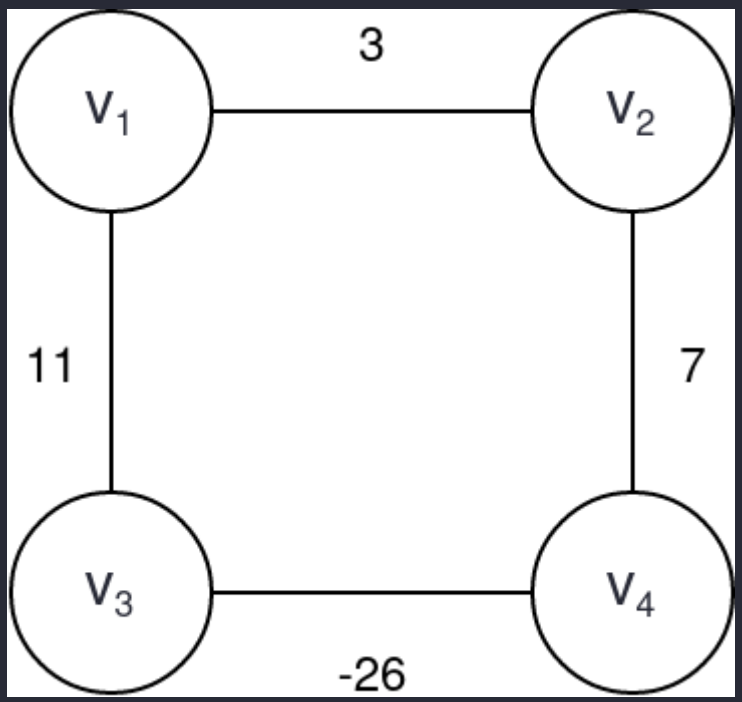
b)

Invalid situation:

For a invalid situation to occur, there needs to be a choice where there is a heavy weight on an edge, large enough to not be favoured until surrounding nodes which could be affected by the negative are become invariant, and then the next node has to have a negative weight which will create a total smaller than the other routes.

Dijkstra's algorithm would make a local decision between the different weighted edges connected to the node but then go down the route with the smaller initial weight, but on a large scale, is smaller if you went down the large weight and then the negative number, however cannot due to making the previous choices invariant. This will create a invalid output due to there being a smaller option possible however was not taken.

Start = V_1

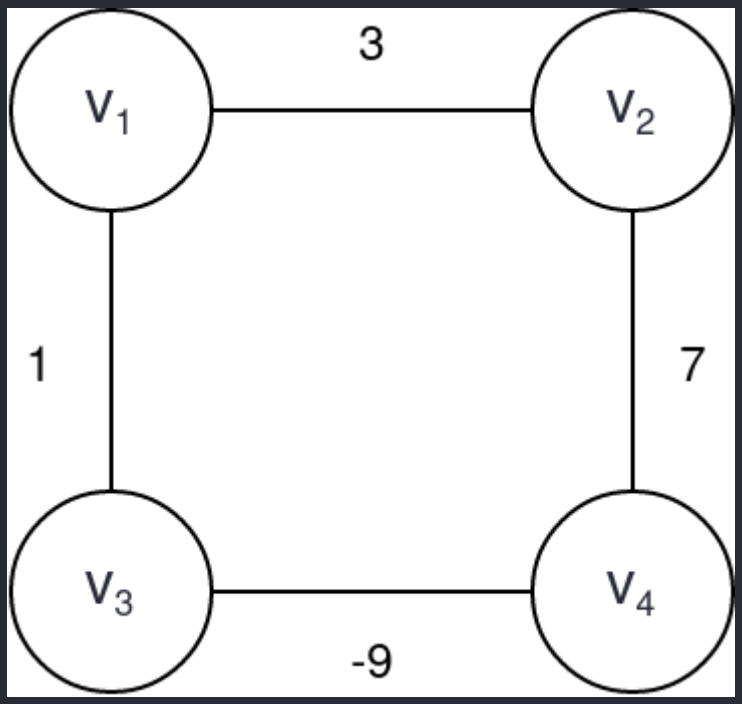


As we chose the smallest and made it an invariant, the node v4 becomes invariant before finding the negative weight which could have made the v4 path total lower than what the invariant is.

Valid:

If the weights before the negative are less than the other weights the negative will be less than all the others and made invariant before the other options which will create a valid graph as dijkstras will go down the lesser path first to then notice the negative weight to compare to the rest of the options and then act on the information it has.

Start = V_1



This will work as V_3 is less than V_2 which causes dijkstra to see there is a negative weight, which is acted upon first rather than the other weights, this will then become an invariant and affect the calculation of the other weights with this new information rather than discovering it last and not being able to change the weights which could be lower but they are invariant.