Table of Contents

Task1:

Task1(a):

A data structure that can store the already arrived busses, print the busses that need the service, and be able to dispatch said busses, is needed.

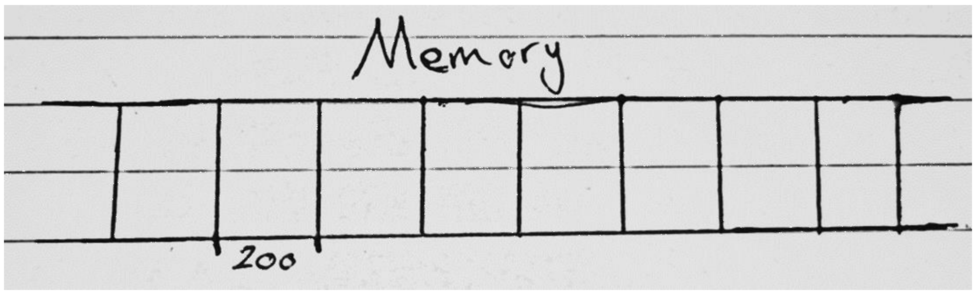
Basic ADT:

- Store unknown amount of already arrived Busses
- Print Busses that need to get the service
- Dispatch finished Busses

The data structure that matches with our requirements is List. And therefore, to maintain time and space complexities, we need a Linked List.

In this system, the busses will be declared one by one, depending on the number of busses that arrive.

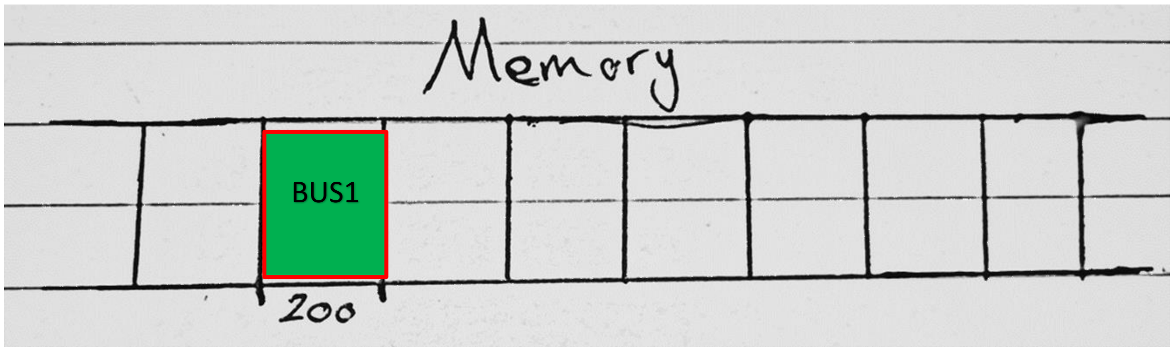
Suppose that we have this memory.



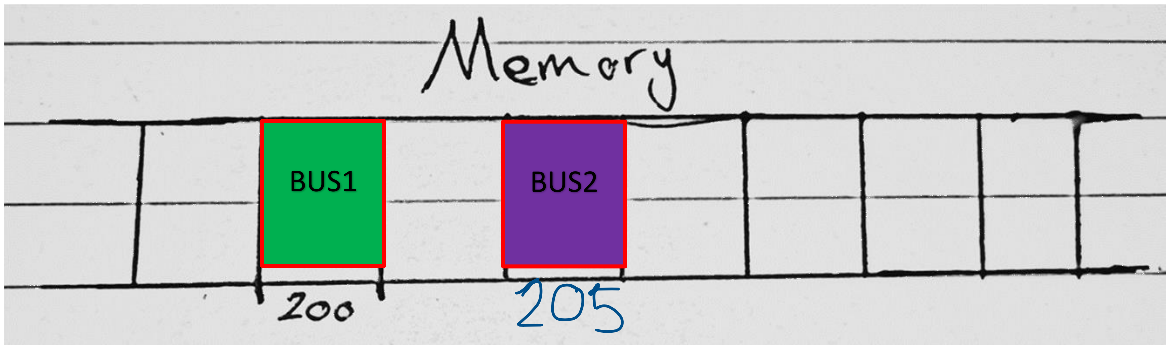
We want to insert some data into it, this data being: BUS1, BUS2, and BUS3.

BUS1 BUS2 BUS3

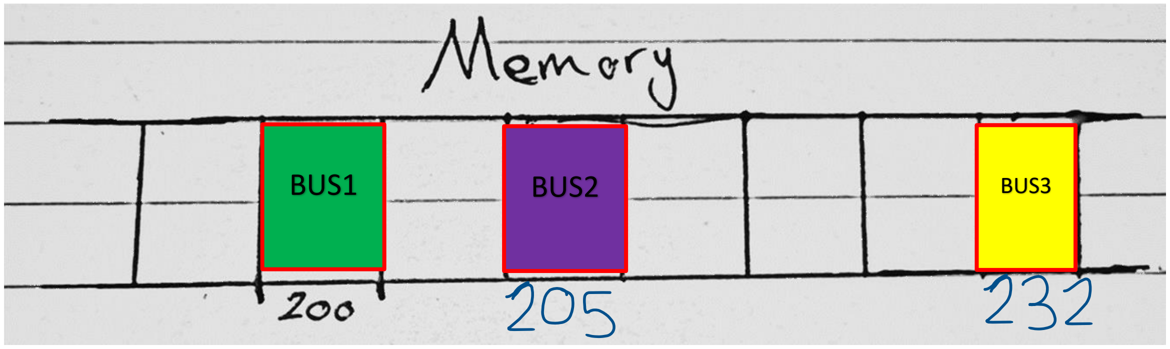
The busses will be stored in the memory not in a contiguous way, but rather scattered as they are not declared separately(non-contiguous). In a normal List, the values are stored in a contiguous way, which has a higher time complexity for certain operations.



As the elements are declared individually in a Linked List, they will not necessarily by after one another.

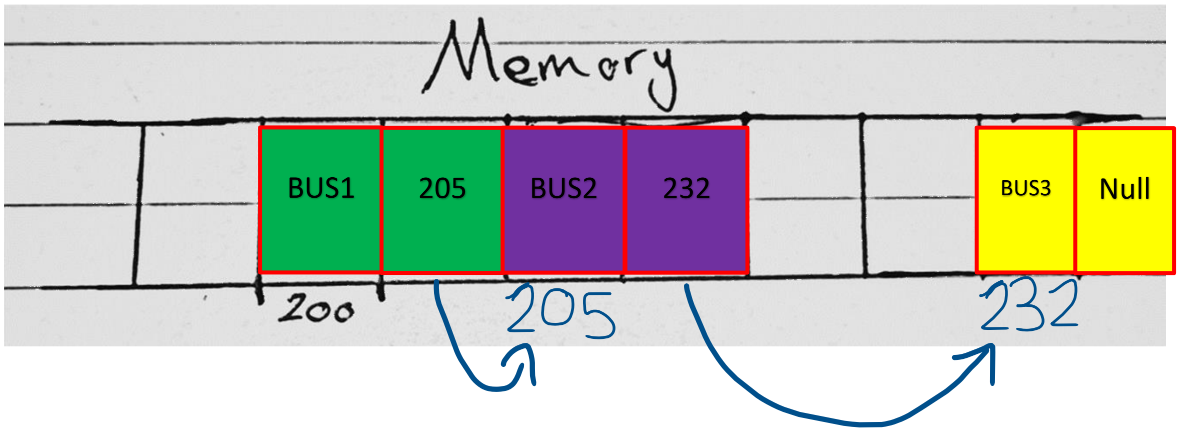


And there is no specific pattern to how they are stored in memory.



But that way, they are not linked with each other, and there is no way to know the sequence of items in the Linked List. Therefore, we need to link them together for them to become a connected list.

In a Linked List, to link the elements together so they are in one list, we store a pointer to the next element from our current element.



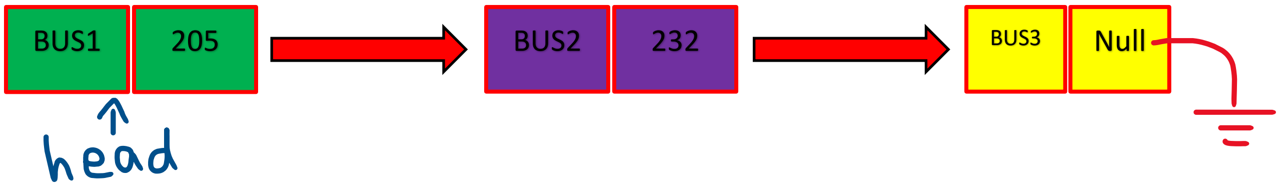
Each element contains two values:

- The data of the element itself. (Bus)
- A pointer to the next element. (Next Bus)

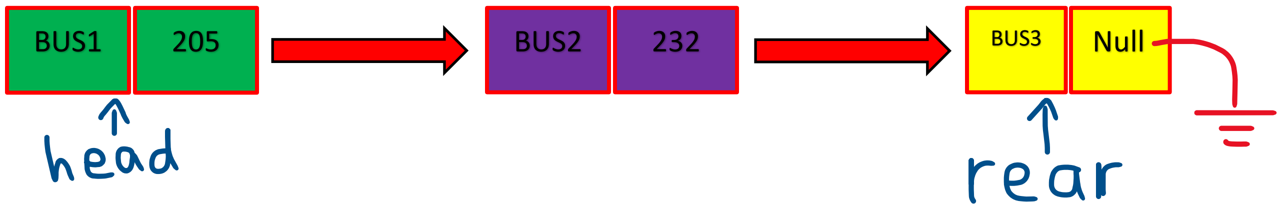
The pointer value of the last element is Null.

Considering our system, let’s call this element a Bus. Each Bus contains data of the Bus, and a pointer to the next Bus.

The first Bus in the list will be called the head, and this is used to traverse the list by using the references of the Busses to reach the next Bus until we reach the last Bus, which has a null reference, indicating that it is the last element.



Usually, the only information always kept in the Linked List is the head Bus, which is used to access the other Busses in the list, and doing various operations through it. Accessing elements in a Linked List has a high time complexity, O(n), in the worst case. Therefore, we can add another piece of information to be stored in the Linked List, and that it is the last element in the List, let’s call it the rear Bus.



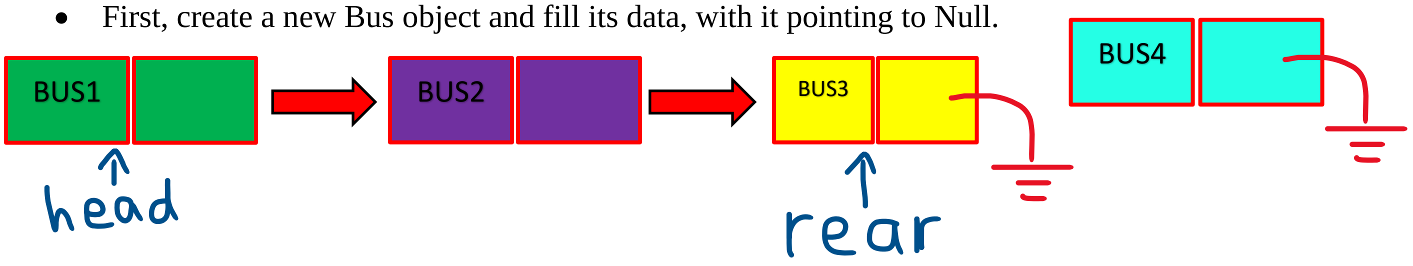
Valid Operations:

— No Busses arrived means head is Null.
— Insert Bus.
— Dispatch Bus (remove).
— Print already arrived Busses.

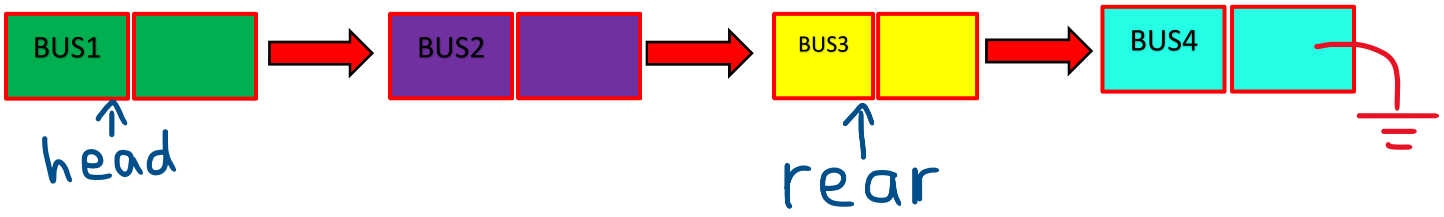
Insert:

When inserting a new Bus to the list:

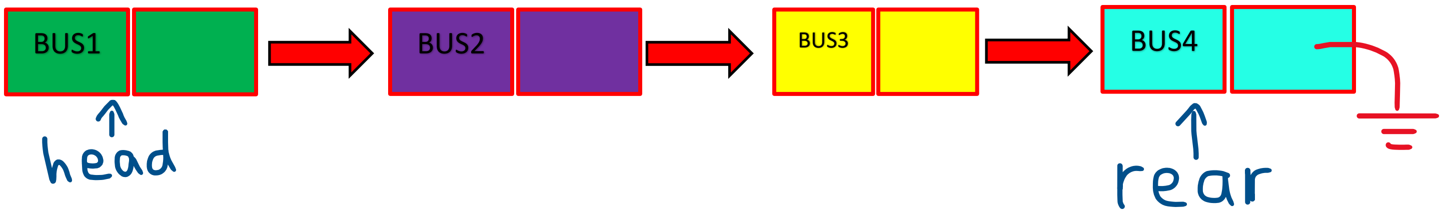
- First, create a new Bus object and fill its data, with it pointing to Null.



- Access the last Bus object in the array and update the reference of it to point to the new Bus.



- The rear Bus in the Linked List should also be updated to be the newly added Bus.



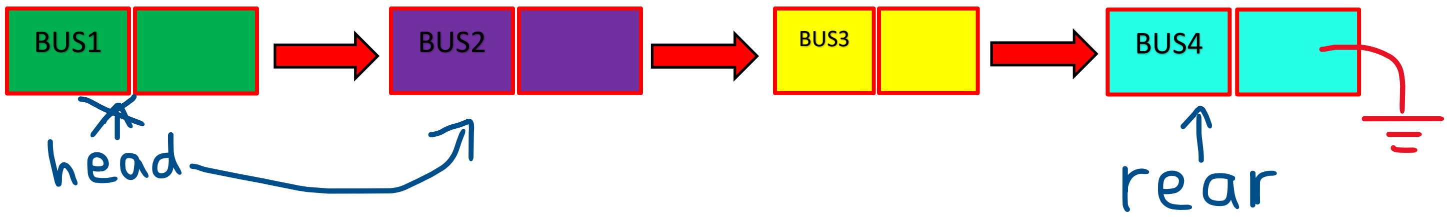
Considering our system, we have no need to insert an item anywhere in the List, only at the end.

The insert operation has a high time complexity as it needs to traverse through all elements to reach the last Bus object, but by using the rear Bus stored in the Linked List, this operation becomes much more efficient in terms of time complexity, as instead of O(n), it becomes O(1).

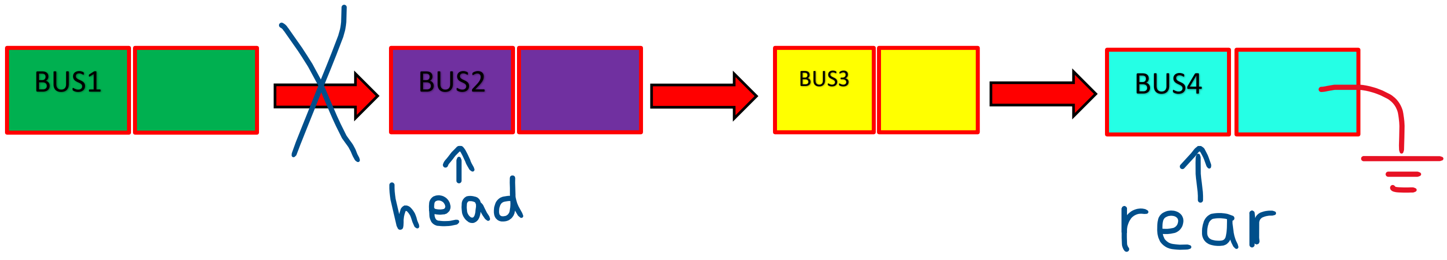
Dispatch (Remove):

When removing a Bus from the list:

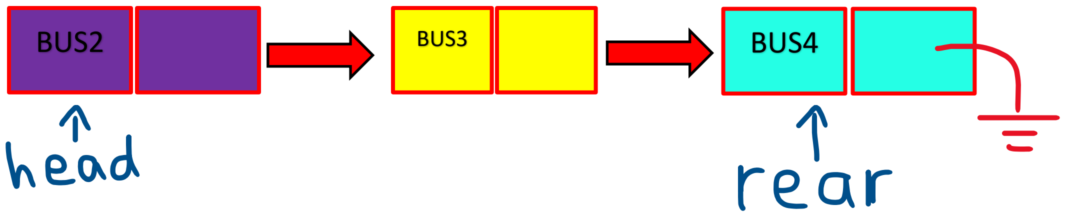
1. To remove the head, we set the new head as next Bus after the head, by using the reference of it in the original head.



2. Update the reference of the new head in the previous head, and set it to Null.



3. This is the final result:



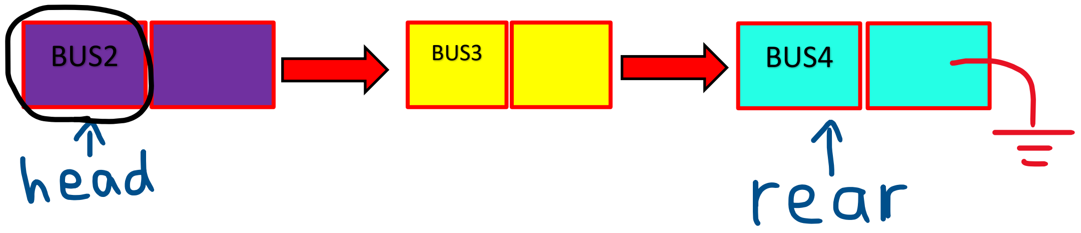
The dispatching of Busses will only happen at the start of the Linked List, meeting the requirements of our system.

This dispatch operation does not have a high time complexity, as we only need operations to update the head of the list. It has a time complexity of O(1).

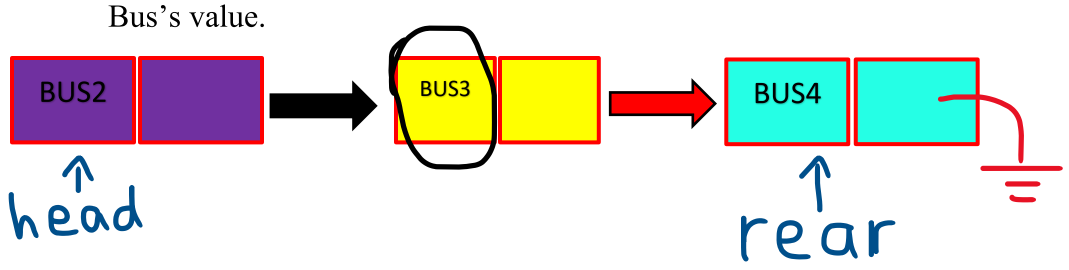
Print:

To print all Busses in the List, we need to traverse the whole list. This process has a bad time complexity of O(n), which depends on the number of Busses in the list.

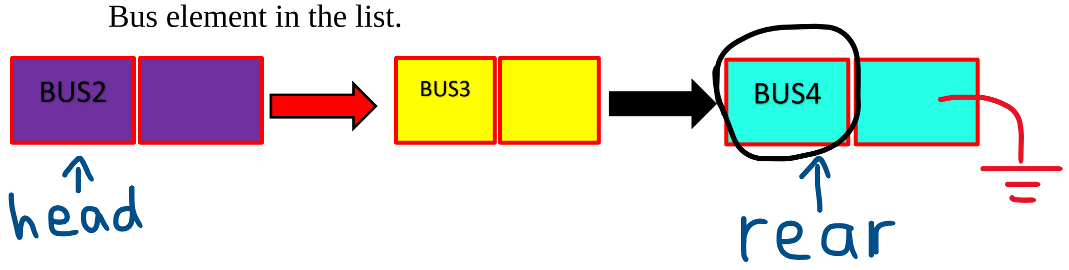
1. Use the head as the first element of the array, and print its value.



2. After printing the first value, use its reference to the next Bus, and then print the next Bus's value.



3. Continue with pattern, going from the current Bus to the next one, until we reach the last Bus element in the list.

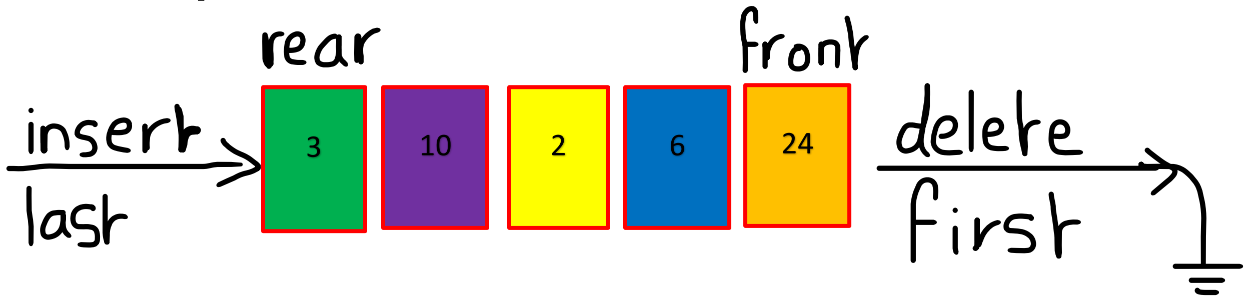


4. Print the value of the last Bus, and then check its reference to the next Bus. If the reference is Null then stop printing the values as there are no more elements after this Bus.

Traversing the Linked List has the worst time complexity our of its operations, as it is always O(n).

Task1(b):

A Queue is a data structure where the first element inserted is the element that is the first to be removed. In it, elements are inserted at the end of the queue, and elements are deleted at the start/head of the queue.



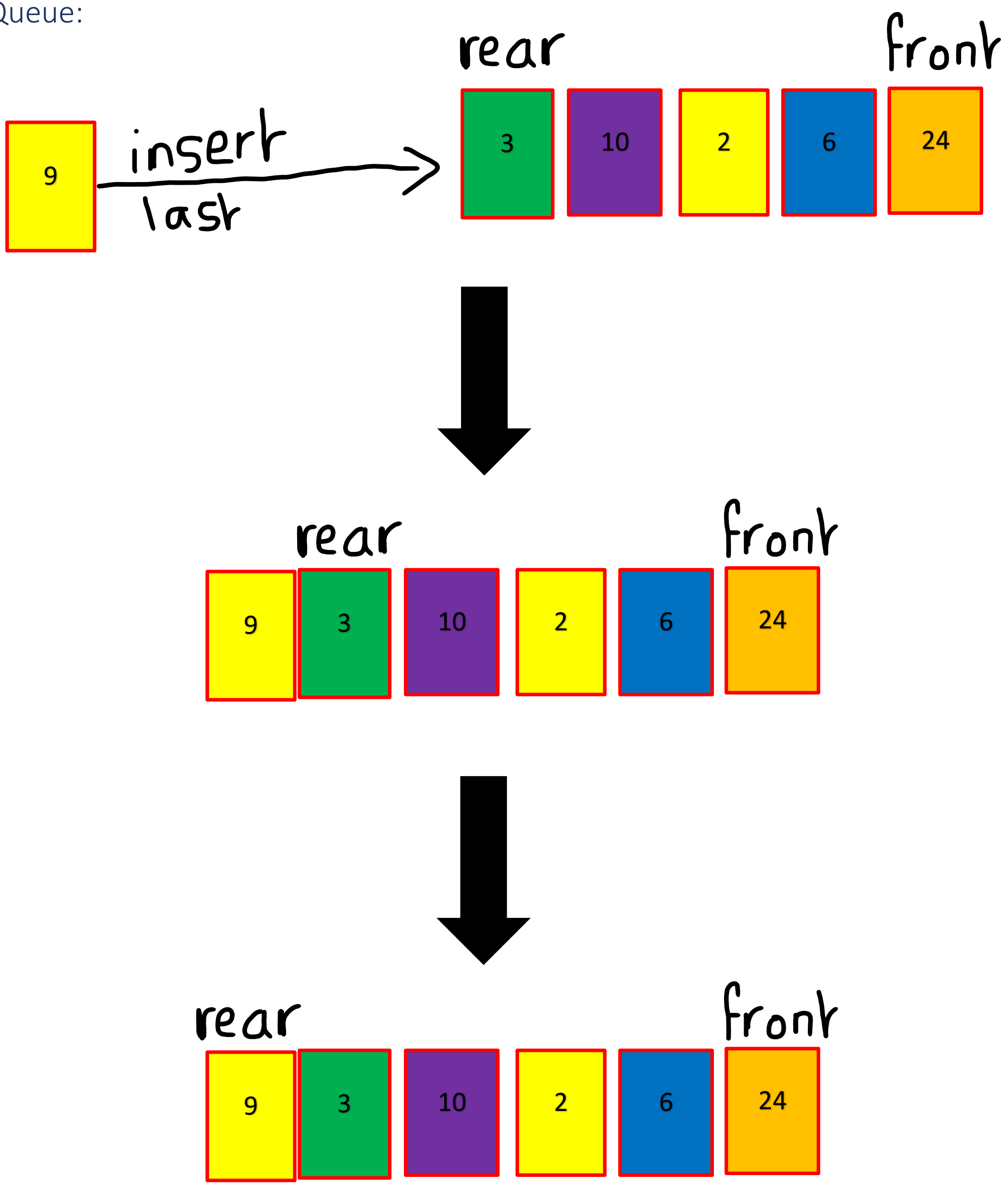
Queue has two main operations:

- Inserting of elements, called EnQueue.
- Removal of elements, called DeQueue.

In this Queue, the EnQueue works by inserting elements to the end/rear of the Queue. Meaning that there is option to insert elements at the start/front of the Queue, or anywhere else in the it.

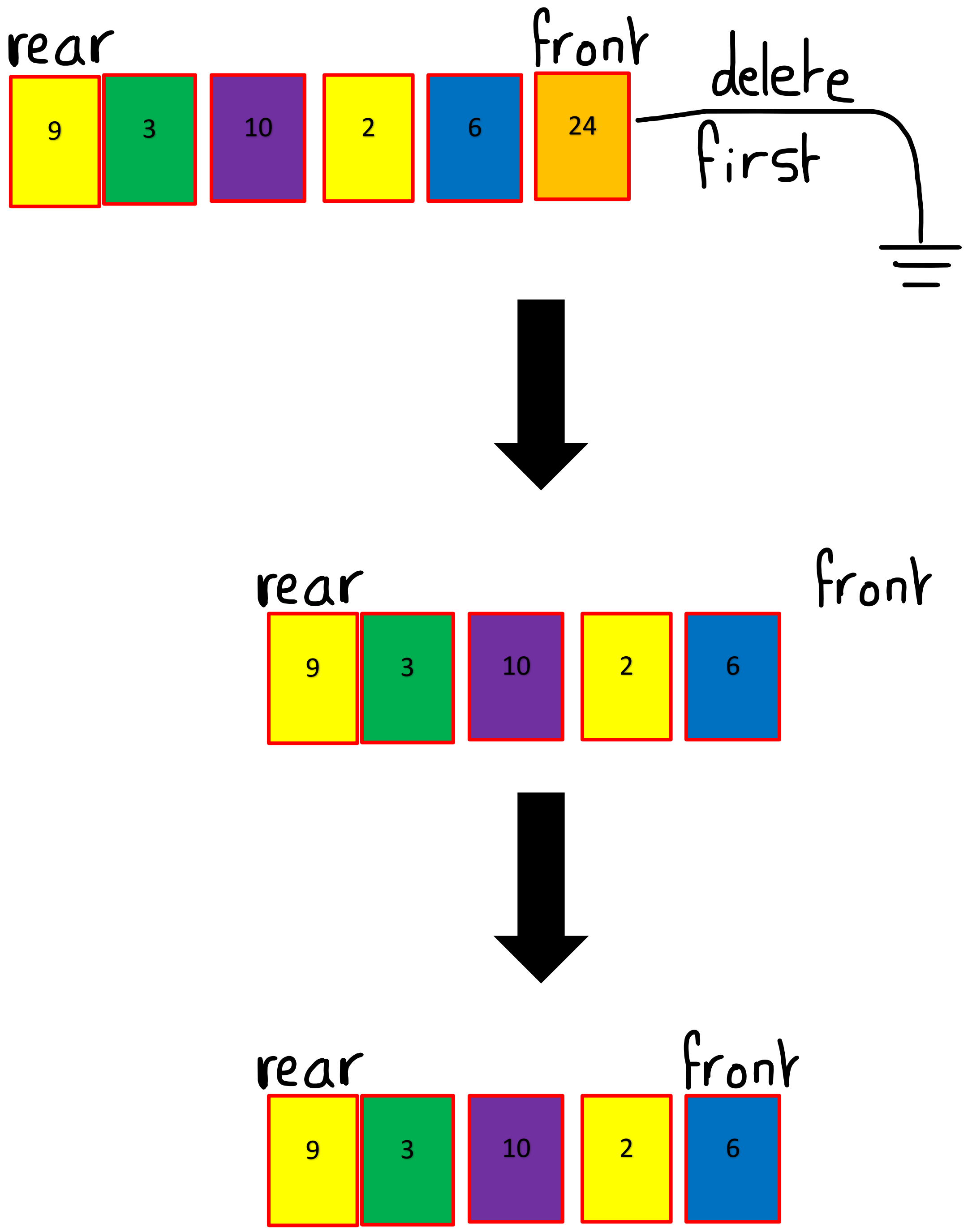
The DeQueue works by removing the element currently at the start/front of the Queue. Meaning that there is no option to delete elements at the end/rear of the Queue, or anywhere else in it.

EnQueue:



As the EnQueue operation only inserts elements at the end of the queue, and we have said rear stored, this operation's time complexity becomes O(1), as in it has a constant runtime.

DeQueue:



As the DeQueue operation only deletes the elements at the front of the queue, and this front's position is stored, this operation's time complexity becomes O(1), as in it has a constant runtime.

Task1(c):

As with almost every concept in the world, we need a scale to measure the effectiveness of an algorithm. When dealing with algorithms, the most common and correct method to measure the effectiveness of said algorithms is to calculate the time needed to accomplish them. The faster they are at outputting correct and valid results, the more efficient and effective an algorithm is considered. The main measurement of these algorithms is by calculating their time complexity.[1]

Time complexity, or in other words, Asymptotic Analysis. It is the computing of running time of any function, method, or operation.[1] The time complexity is calculated by how many times a piece of code is repeated depending on a certain input, therefore, it is not a definite value, but a variable that can change depending on many factors such as different computers, different processors, and others.[18]

To know how asymptotic analysis is used to measure the effectiveness and efficiency of an algorithm, take the following example:

We have two algorithms used to find whether a number is a prime number or not. The first algorithm runs the code (n-1) times, where n is the input number. While the second algorithm runs the code ($\sqrt{n}$) times, where n is the input number. For small inputs, the time taken to find the whether a number is a prime or not won't differ by much, but when the input gets bigger, huge differences in the time taken between both algorithms can be found. For example, if the input number is 1,000,000, the first algorithm will run the code (1,000,000 - 1), which is 999,999, times to find whether the number 1,000,000 is a prime number or not. But the second algorithm will take ($\sqrt{1{,}000{,}000}$), which 1,000, times to find whether 1,000,000 is prime number. It can be seen that there is a very huge difference between the number of times each algorithm will take to find the result, where the first algorithm took 1,000,000 times, while the second only took 1,000 times. This means that the second algorithm is much more effective, efficient, and faster than the first algorithm, as it takes much less time to run than the first, and this difference will only grow for bigger inputs.[19]

In other words, the time taken to perform the first algorithm(n-1) increases linearly with the input n, while the time taken to perform the second algorithm($\sqrt{n}$) increases logarithmically with the input n.[19]

To find the better algorithm, with the best effectiveness and efficiency for every situation, we need to simulate the time they need. Asymptotic analysis helps very much in this process as it directly gives an approximate value of how many times an algorithm runs its code. It also helps in creating faster and more efficient programs by choosing the best algorithms, with the best time complexity.[18]

Using asymptotic analysis, the exact time is not important. For example, O(5n) becomes O(n) with asymptotic analysis. This is because knowing that an operation has a time complexity of O(5n) is not of much significance compared to knowing that the operation has a linear runtime O(n).[19]

Some Asymptotic analysis rules:[19]

• 5 * O(1) = O(1)	Constant runtime
• n * O(1) = O(n)	Linear runtime
• 5 * O(n) = O(n)	Linear runtime
• O(1) + O(1) + 5 = O(1)	Constant runtime
• n * O(n) = O(n^2)	Quadratic runtime
• 9 * O($\log n$) = O($\log n$)	Logarithmic runtime
• n^2 * O(n) = O(n^3)	Cubic runtime

Task1(d):

To determine the efficiency and effectiveness of an algorithm, the two main measures of time complexity and space complexity are used.[2] There is no way to compare both of them directly, and therefore, both are considered for certain situations.[2] For example, a programmer may conclude that using more memory for an operation, but getting results faster, is more ideal for the current program, and therefore use it, but he also may conclude that it is better to use less memory, but with slower results. It differs based on the operation, the program, and the programmer's decision.

For example, the two search algorithms – Linear and Binary search:[18]

Linear Search	Binary Search

`def linearSearch(a, x):`
`  for i in range(0, len(a)):` → $O(len(a))$
`    if a[i] == x:` → O(1)
`      return i` → O(1)
`  return None` → O(1)
	`def binarySearch(a, x):`
	`  start = 0` → O(1)
	`  end = len(a) - 1` → O(1)
	
	`  while start <= end:` → $O(\log len(a))$
	`    mid = (start + end) // 2` → O(1)
	`    if a[mid] == x:` → O(1)
	`      return mid` → O(1)
	`    elif a[mid] > x:` → O(1)
	`      end = mid - 1` → O(1)
	`    else:`
	`      start = mid + 1` → O(1)
	`  return None` → O(1)

In Linear search, as we have to access all elements and check them all to find the index, the time complexity is O(n) in the worst-case scenario, where n is the number of elements in the array. Moreover, the space complexity is n, where n is the number of elements in the array.[19]

Whereas in Binary search, we don't need to access all the elements in the array, and by reducing the number of items that we need to access by half every iteration, the time complexity becomes $O(\log n)$ in the worst-case scenario – much better than the time complexity of Linear search – where n is the number of elements in the array. The space complexity for this algorithm is the same as Linear search(n), where n is the number elements in the array. But, this algorithm has the condition that for it to work properly, the array given to it needs to be sorted.[19]

Given the results, the Binary search algorithm is more efficient than the Linear search algorithm, given that the array is sorted, as even though both have the same space complexity, the time complexity of both differs greatly. For example, if the size of the array is 1,000,000, then it is looped that many times in Linear search. On the other hand, it will only be looped about 20 times in Binary search if the array is sorted. A huge difference in efficiency.[19]

Task1(e):

Obtaining something by sacrificing another thing, that is what a trade-off usually means. A trade-off in ADT has two different meanings, considering the methods of determining efficiency and effectiveness of an algorithm:

- The first type is by doing an operation in less time, but by using more memory. In other words, less time complexity, but higher space complexity.[3]
- The second type is by doing an operation using less memory, but in more time. In other words, higher time complexity, but less space complexity.[3]

An example of this trade-off is of uncompressed and compressed data in memory.[3] Storing data uncompressed uses more space in memory, meaning more space complexity, but takes less time to return with said data, meaning less time complexity. While on the other hand, storing data compressed uses less space, meaning less space complexity, but takes more time to decompress the data and return it.[4]

In the example above, it is shown how the storing compressed and uncompressed data can have multiple ways of being done, depending on the trade-off in it. The best result would be for an algorithm to use less memory with less time, but this is not always possible, and therefore, a trade-off may be used.[4]

The choice of which trade-off to choose depends on multiple factors such as the programmer, the program, the function that uses the algorithm, and others. The programmer may want to save more memory as they do not have much, so they prioritize algorithms that use less memory, And they may also want to make an operation faster, and are willing to use more memory for it, so they prioritize algorithms that take less time.[18]

For example, let's take the Dijkstra's algorithm example in Task 2(a). In that example, we used an adjacency matrix as the representation for the graph. This representation has a space complexity of v^2, where v is the number of vertices in the graph. The trade-off in this is that it has a better time complexity compared to using other graph representation, although it uses much more space complexity. In this case, if we were to use the adjacency list, for denser graphs, the algorithm would have more time complexity, because it would then need more processes to reach certain information needed for it to find the shortest path, as not all vertices are necessarily connected to each other. On the other hand, the currently used adjacency matrix uses more space complexity, to trade for better time complexity, compared to the other graph representations. Therefore, the time complexity of this algorithm can differ depending on the space complexity of the graph representation used. Time complexity may increase with less space complexity, and it may also decrease with more space complexity.

Task1(f):

There are mainly two ways of representing data structures, and they are:[19]

- Mathematical and Logical models, or in other words, Abstract Data Type(ADT).
- Implementation.

The mathematical and logical models are only abstract views of data structures that tell what features and operations make up a data structure.[19]

The implementation can be done in many ways, and there can be many ways to implement an abstract data type. It is the actual data structure that the programmer creates. Data structures that only have an ADT without implementation are called implementation independent data structures.[18]

The advantages that separating these ways into two to create implementation independent data structures are three. These advantages being:

1. The first advantage is the efficient use of memory. Through implementation independent data structures, the use of memory can be optimized depending on the requirements and how the data structure is going to be used.[5] This is related to space complexity, and the less complexity there is, the better the data structure is. It gives us a plan on how to proceed with the implementation, instead of starting with the implementation, and then needing to improve upon it.[18]
2. The second advantage is the ability to reuse the data structure. Implementation independent data structures can be implemented in many ways, and therefore can be reused many times, in many different situations.[6] Different situations demand different problem-solving approaches, which may need different implementations for them, but it can still be the same implementation independent data structure.[18] Said data structures can even be turned into libraries, which can be used by many people if allowed for public use.[6]
3. The last advantage is abstraction.[6] Implementation independent data structures can be utilized in many ways, and therefore other data structures can be built upon them. This is called abstraction. For example, using the Linked List ADT, a Queue can be built upon it to create a Queue with Linked List ADT, which has its own benefits, combining both. Abstraction also means that data structure would be independent on an implementation, and is subject to improvement, based on different system, without an implementation.[18]

Task1(g):

Selection Sort VS. Merge Sort(Recursive)

Selection Sort

```
selection(a)
{
    n = length of array a → O(1)
    i = 0 → O(1)
    while i <= (n - 1) → (n – 1) * O(n) → $O(n^2)$
    {
        min = i → O(1)
        j = i + 1 → O(1)
        while j <= n → O(n)
        {
            if a[j] < a[min] → O(1)
            {
                min = j → O(1)
            }
            j = j + 1 → O(1)
        }

        temp = a[i] → O(1)
        a[i] = a[min] → O(1)
        a[min] = temp → O(1)
        i = i + 1 → O(1)
    }
}
```

Merge Sort(Recursive)

```
merge(a) {
    n = length of array a → O(1)
    if n < 2 then abort merge function → O(1)
    mid = n / 2 → O(1)
    left = array with (mid) size → O(mid)
    right = array with (n - mid) size → O(n - mid)
    i = 0 → O(1)
    while i <= (mid - 1) → O(mid – 1) → O(mid)
    {
        left[i] = a[i] → O(1)
        i = i + 1 → O(1)
    }
    i = mid → O(1)
    while i <= (n - 1) → O((n – 1) – mid) → O(n)
    {
        right[i - mid] = a[i] → O(1)
        i = i + 1 → O(1)
    }
    merge(left) → $O(mid * \log mid)$
    merge(right) → $O((n - mid) * \log(n - mid))$ → $O(n * \log n)$
    sort(left, right, a) → O(n)
}
sort(left, right, a) {
    Ln = length of array left → O(1)
    Rn = length of array right → O(1)
    i = j = k = 0 → O(1)
    while i < Ln AND j < Rn → O(Ln + Rn) → O(size of a)
    {
        if left[i] <= right[j] → O(1)
        {
            a[k] = left[i] → O(1)
            i = i + 1 → O(1)
        }
        else
        {
            a[k] = right[j] → O(1)
            j = j + 1 → O(1)
        }
        k = k + 1 → O(1)
    }
    while i < Ln { → O(Ln)
        a[k] = left[i] → O(1)
        i = i + 1 → O(1)
        k = k + 1 → O(1)
    }
    while j < Rn { → O(Rn)
        a[k] = right[j] → O(1)
        j = j + 1 → O(1)
        k = k + 1 → O(1)
    }
}
```

The time complexity in Selection Sort is O(n^2), as there is a nested loop. Moreover, its space complexity is O(1), meaning there is only one useless space used other than the space of the array itself.[10]

The Selection Sort is a very efficient algorithm considering the space complexity alone, but is not so efficient in terms of time complexity.

On the other hand, the time complexity in Merge Sort is O(n ($\log n$)). Additionally, its space complexity is O(n), because it uses space in memory other than what is needed to maintain the array itself.[11]

The Merge Sort is much better than Selection Sort in terms of time complexity, although it has a higher space complexity.

The approach of Selection Sort is that it iterates through all elements of a list, which has an n size, n times.

Whereas the approach in Merge Sort is that the given list is split in half, and these halves are split further and so on, until we have very small lists, then these small lists get sorted. After that, the lists are combined together by taking the smallest of each pair and sorting them in the original list, till finally, we get the list, which originally had an n size, sorted.[18]

The Merge Sort is a recursive algorithm, and therefore the time complexity can be measured using the Master Theorem. The general form of the Master Theorem is:

$T(n) = a * T(n - k) + f(n)$ where a is the number of recursions in the same function, and f(n) is the time complexity of the other operations.

The Merge Sort is generally calculated in the following way:

$T(n) = 2T\left(\frac{n}{2}\right) + n$ and $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$ Therefore:

$T(n) = 4T\left(\frac{n}{4}\right) + 2n$ And:

$T(n) = 8T\left(\frac{n}{8}\right) + 3n$

We can then deduce that it is represented in the following way: $T(n) = 2^k T\left(\frac{n}{2^k}\right) + k * n$

As we know that $T(1) = 1$, we can do this: $\frac{n}{2^k} = 1 \rightarrow n = 2^k$ And $\log n = k$

$T(n) = n * T(1) + \log(n) * n = n * 1 + \log(n) * n$

Because the Big O Notation takes the worst-case scenario, it becomes like this:

$O(n) = \text{n} * \log(n)$

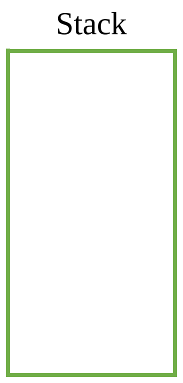
Task1(h):

A stack is a first-in-last-out (FILO) data structure. One of its implementations is the memory stack, where all local variables and function calls are stored and executed. There are different operations a stack is capable of doing that a memory stack can also do:[18]

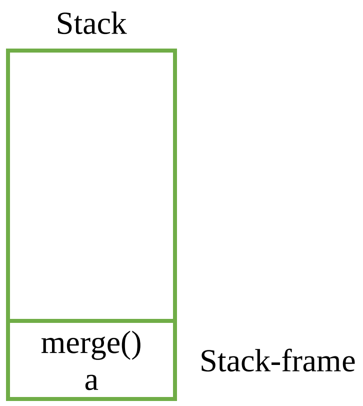
- Push: function calls are pushed to the memory stack. The local variables of the function called are also stored with it. This is to have a reference to these variables to call them, and not need to initialize them every time, inside the same function call.[18]
- Pop: once functions finish executing, they are popped(removed) from the stack, and the previous function in the stack continues. The function that called the function that finished is the one that continues executing. When popping a function call, the local variables stored with it are also removed from the stack. This is to save memory in the stack, and not use the memory to store variable that most likely will not be used later.[18]
- Stack overflow: if the stack is filled with operations, then a stack overflow happens and the program crashes.[18]

For example, considering the merge sort algorithm pseudo code above, the memory stack would behave like this:[18] [19]

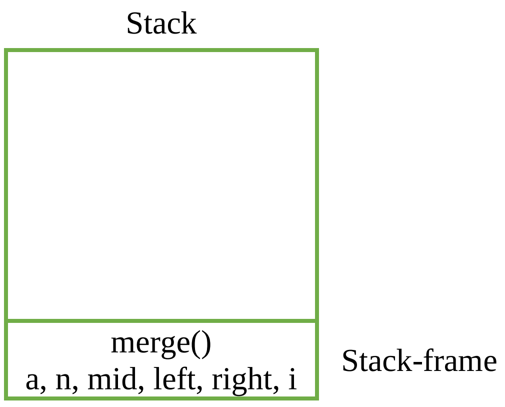
1. When the program first starts, the memory stack would look similar to this:

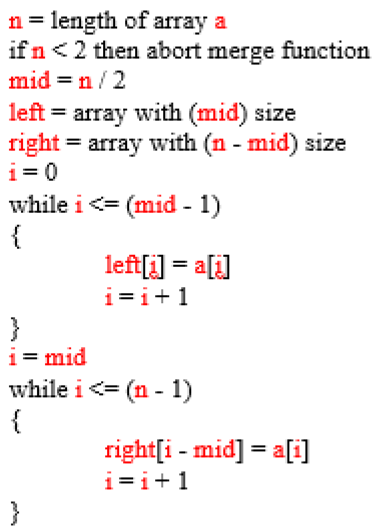


2. When the merge(a) function is called, the memory stack would create the stack frame as the first function:



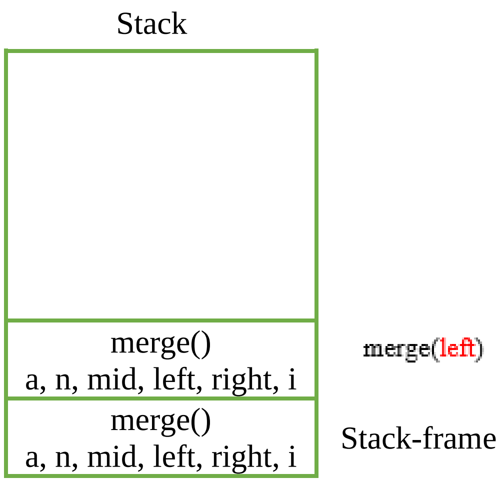
3. Then as the algorithm is executed, the local variables in the merge(a) function are stored in the stack frame with the function:



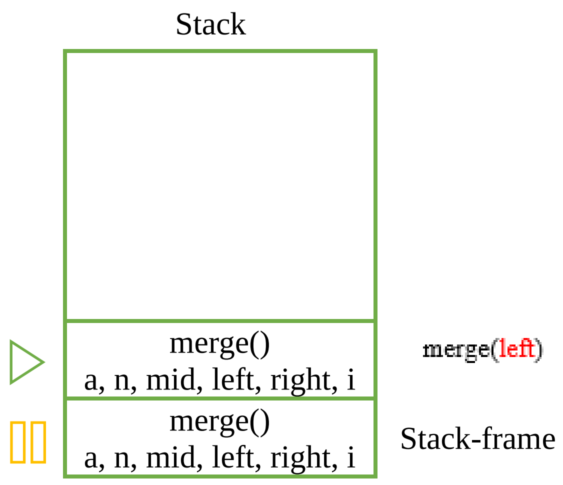


```
n = length of array a
if n < 2 then abort merge function
mid = n / 2
left = array with (mid) size
right = array with (n - mid) size
i = 0
while i <= (mid - 1)
{
        left[i] = a[i]
        i = i + 1
}
i = mid
while i <= (n - 1)
{
        right[i - mid] = a[i]
        i = i + 1
}
```

4. When the merge(a) function recursively calls itself again, this new merge(a) is added to the stack:

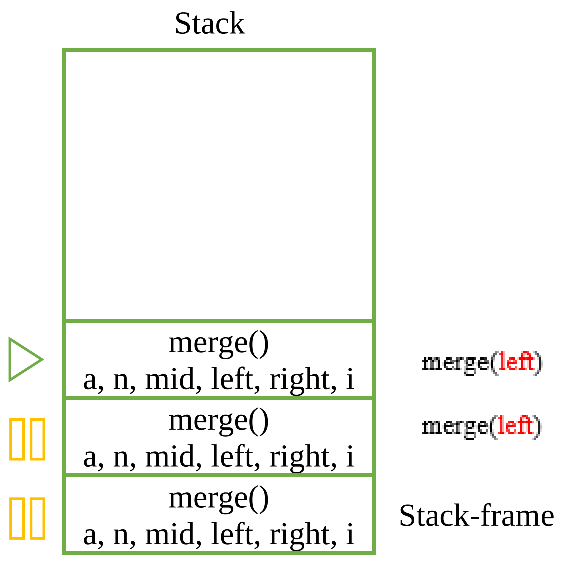


5. While the second merge() is being executed, the other merge() function at the bottom of the stack is paused:

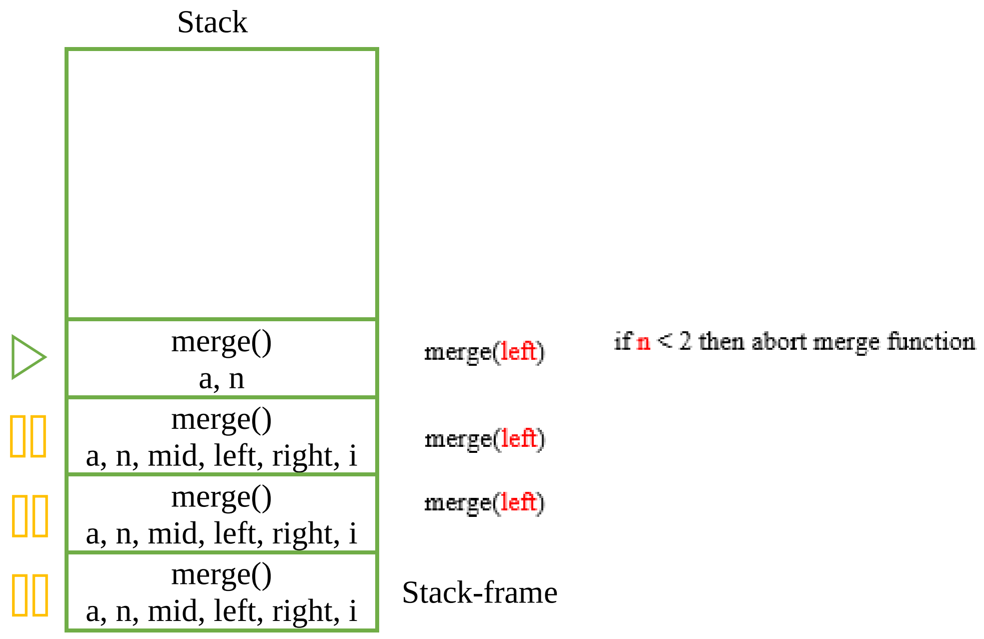


```
n = length of array a
if n < 2 then abort merge function
mid = n / 2
left = array with (mid) size
right = array with (n - mid) size
i = 0
while i <= (mid - 1)
{
            left[i] = a[i]
            i = i + 1
}
i = mid
while i <= (n - 1)
{
            right[i - mid] = a[i]
            i = i + 1
}
```

6. Then merge() function calls itself again, and is added to the memory stack again. And while the newly added merge() function is being executed, the other function calls in the memory stack are paused:

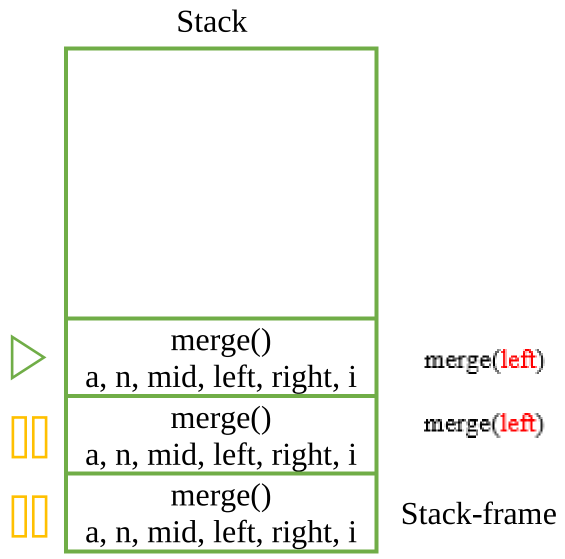

Stack
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge(left)
merge(left)
Stack-frame

7. The previous process is being continuously done until the left array becomes so small that the function is aborted:



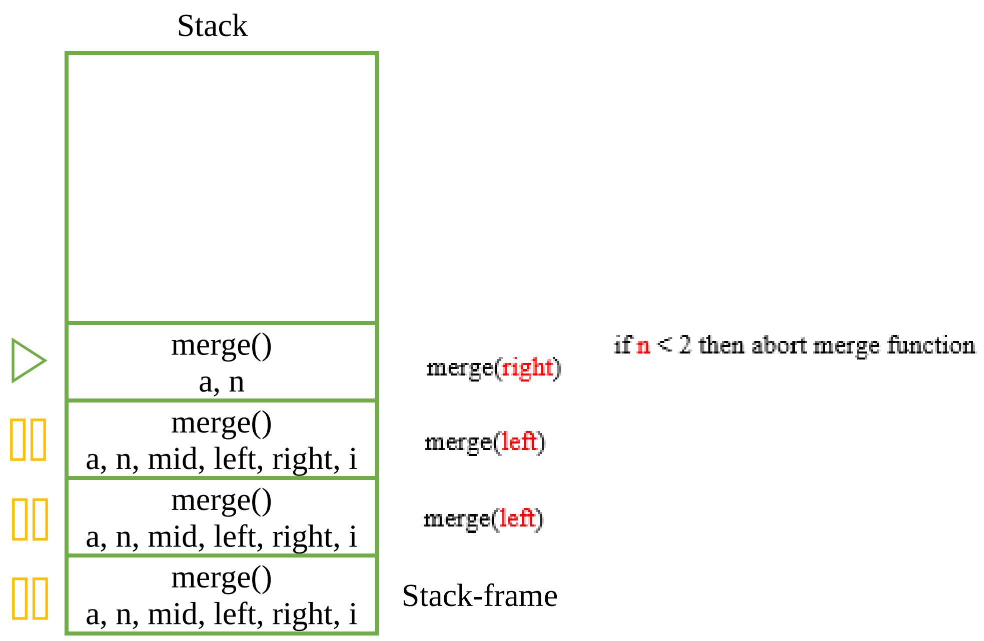
Stack
merge()
a, n
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge(left)
if n < 2 then abort merge function
merge(left)
merge(left)
Stack-frame

8. Once the merge() function is aborted, it is removed(popped) from the stack, and the previous function call is executed:

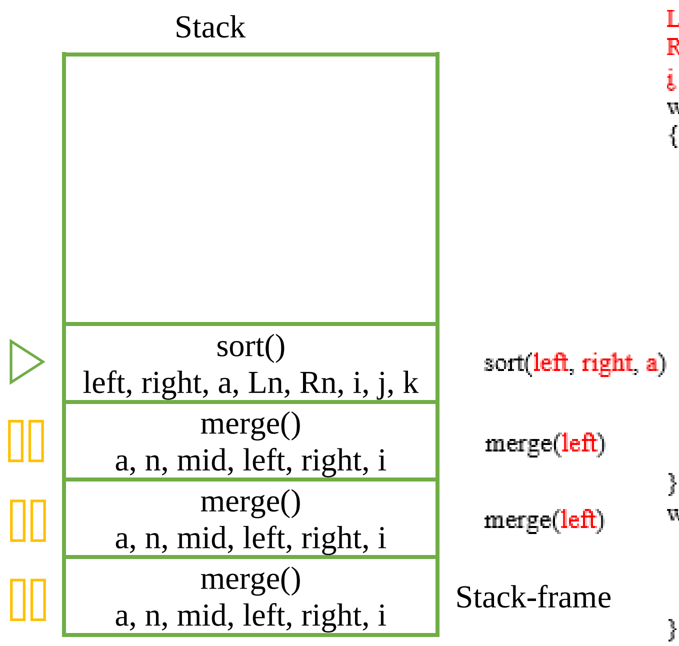


Stack
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge(left)
merge(left)
Stack-frame

9. After that, the next recursive function call is done, in which another function call is added to the stack, but is then aborted for the same reason the previous call was popped from the stack; the right array is very small:

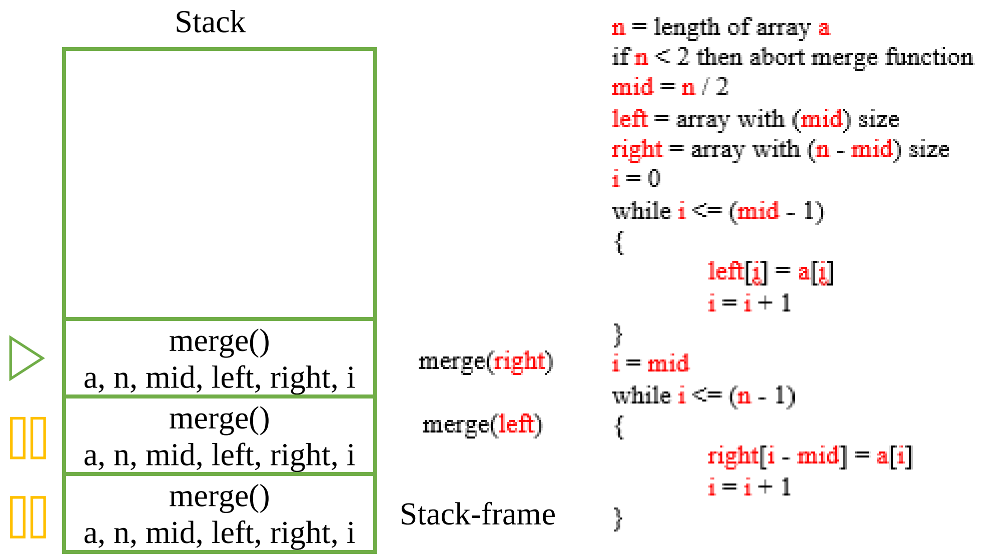

Stack
merge()
a, n
merge(right)
if n < 2 then abort merge function
merge()
a, n, mid, left, right, i
merge(left)
merge()
a, n, mid, left, right, i
merge(left)
merge()
a, n, mid, left, right, i
Stack-frame

10. After that, the next function call is done, but this call is not done recursively, but to another function that sorts the array passed to it:


Stack
sort()
left, right, a, Ln, Rn, i, j, k
sort(left, right, a)
merge()
a, n, mid, left, right, i
merge(left)
merge()
a, n, mid, left, right, i
merge(left)
merge()
a, n, mid, left, right, i
Stack-frame

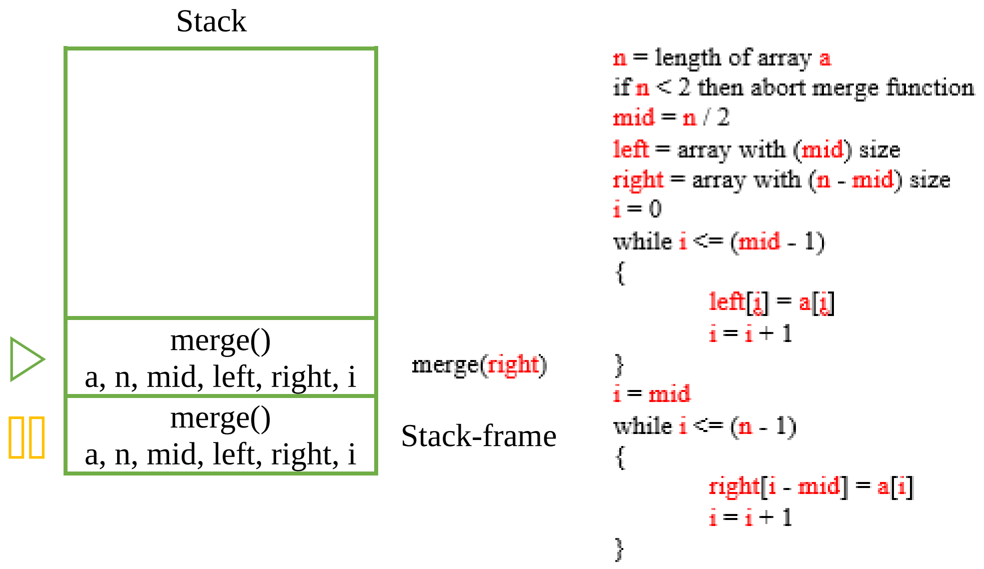
```
Ln = length of array left
Rn = length of array right
i = j = k = 0
while i < Ln AND j < Rn
{
          if left[i] <= right[j]
          {
                    a[k] = left[i]
                    i = i + 1
          }
          else
          {
                    a[k] = right[j]
                    j = j + 1
          }
          k = k + 1
}
while i < Ln {
          a[k] = left[i]
          i = i + 1
          k = k + 1
}
while j < Rn {
          a[k] = right[j]
          j = j + 1
          k = k + 1
}
```

11. Similar to before, when this function finishes executing, it is popped from the stack, and the function that called it is resumed:


Stack
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge()
a, n, mid, left, right, i
merge(right)
merge(left)
Stack-frame

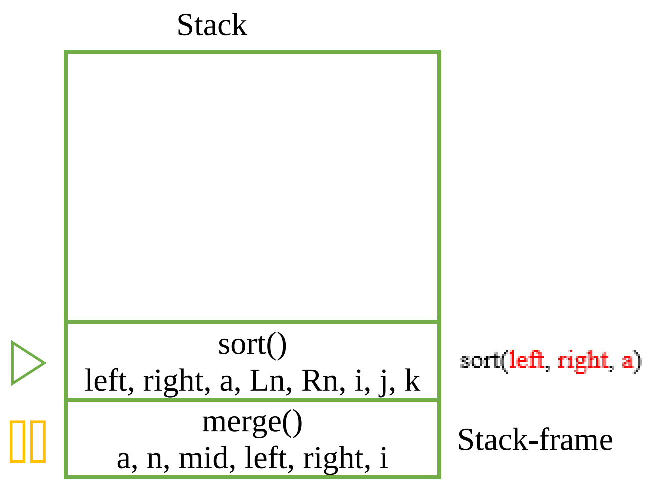
```
n = length of array a
if n < 2 then abort merge function
mid = n / 2
left = array with (mid) size
right = array with (n - mid) size
i = 0
while i <= (mid - 1)
{
            left[i] = a[i]
            i = i + 1
}
i = mid
while i <= (n - 1)
{
            right[i - mid] = a[i]
            i = i + 1
}
```

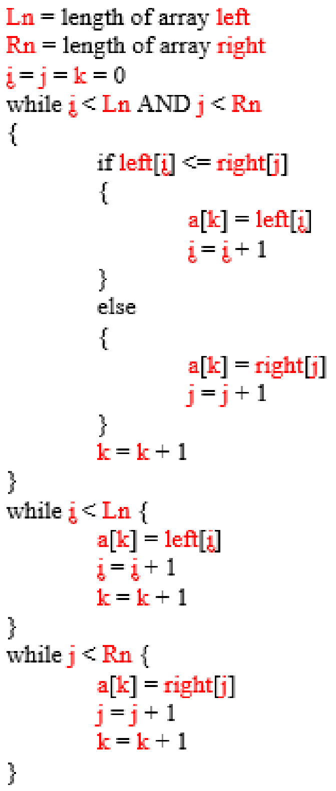
12. This process is repeated until the function's processes and the function calls inside it are finished. After all those are done, the function call is popped from the stack so that its parent function continues executing:



```
n = length of array a
if n < 2 then abort merge function
mid = n / 2
left = array with (mid) size
right = array with (n - mid) size
i = 0
while i <= (mid - 1)
{
            left[i] = a[i]
            i = i + 1
}
i = mid
while i <= (n - 1)
{
            right[i - mid] = a[i]
            i = i + 1
}
```

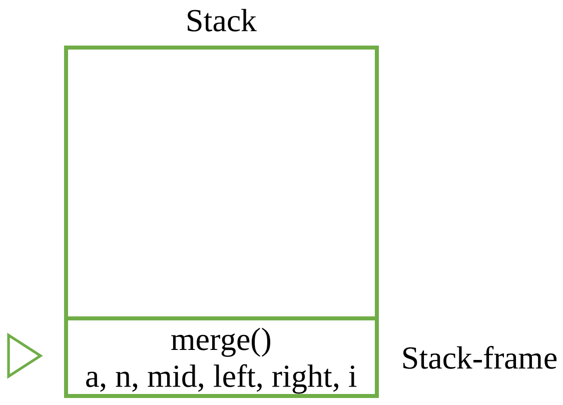
13. This cycle is continued until the last function call of the first merge() function call is executed:



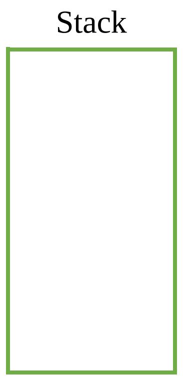


```
Ln = length of array left
Rn = length of array right
i = j = k = 0
while i < Ln AND j < Rn
{
        if left[i] <= right[j]
        {
                a[k] = left[i]
                i = i + 1
        }
        else
        {
                a[k] = right[j]
                j = j + 1
        }
        k = k + 1
}
while i < Ln {
        a[k] = left[i]
        i = i + 1
        k = k + 1
}
while j < Rn {
        a[k] = right[j]
        j = j + 1
        k = k + 1
}
```

14. Then the first merge() function call continues executing:



15. After the first merge() function call is finished, the stack is emptied, and the program finishes:



Task1(i):

Code is provided in ***task1(i).py*** submitted file.

Task1(j):

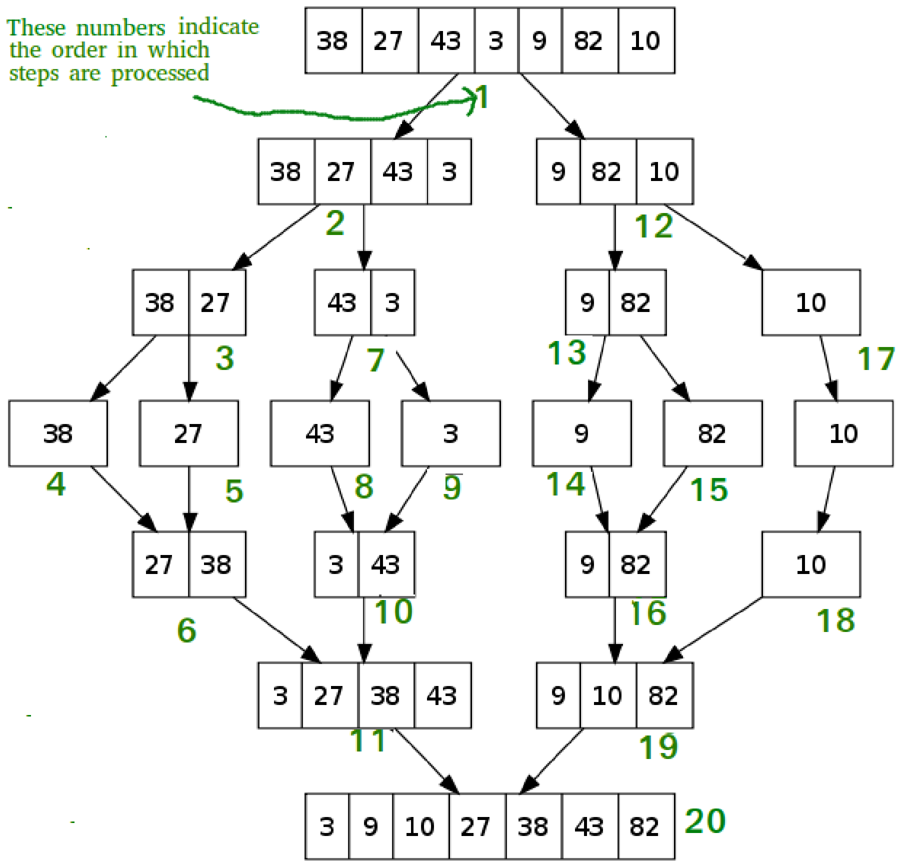
Formal Definition:

Merge Sort(recursive) is a recursive divide and conquer sorting algorithm, used to produce a sorted list by continuously splitting the list into halves, left and right, and then merging them together after sorting them. To sort each half, the algorithm's function keeps recursively calling itself, until it is small enough to sort them while merging.[19]

Logical View:

- Splits the arrays into halves, sorts the data inside these halves, then merges them together.
- Can accept negative values.
- Allows odd array sizes.

12



Valid Operations:

1. split(a): splits array (a) into two halves, left and right, by using the size of the array divided by 2.[18]
2. sort(a, b, c): a is the left half, and b is the right half of the array. c is the array a and b were split from. This operation sorts the array c, by comparing arrays a and b to find the smallest element, and then saving it in array c, in a continuously increasing index.[18]

NOTE: The merge operation is considered as part of the sort operation, as they are done in the same function. It could also be separated as its own operation.

Task1(k):

Encapsulation is the act of wrapping the data, to not allow the user to access it, as it only benefits in the processing of data in the background, away from the implementation.[7]

Information hiding is similar to encapsulation, but it differs in the fact that some users, with certain privileges, are allowed to access said data.

In an ADT, it is important to decide which users will be able to use certain operation, and who will not be able to. Also, it is important to protect some data and hide it form the user as it is only used in the background, not something the user is allowed to access directly.

For example, if in the ADT it is not specified which users are able to use the delete operation, then all users would become capable of using the delete operation, and can thus delete many things that they should not. Consequently, this can potentially destroy the system, and is not very desirable.

Encapsulation and information hiding provide the advantage of allowing the programmer to choose who can access certain data and operations. This means that the data is well protected from outside interference, and allows us to validate who can access which data and operations.[8]

Encapsulation also ensures that an ADT is independent and does not need the interference of other data structures or implementations, and provides better abstraction of the ADT. Abstraction makes it easier to reuse an ADT, so using it in multiple situations, for many data structures, becomes possible.[9]

The data in an ADT is encapsulated, and have information hiding to allow the ADT to be implemented in many different ways. This is possible because an ADT is not an implementation itself, but only includes what an implementation should have. Any changes to any of the different implementations do not affect ADT.[18]

An ADT only specifies a set of inputs and outputs for different operations and functions of the ADT.[15] This is done to hide to the details of the ADT from the implementation, as the implementation does not need a detailed documentation, because it can be done through different approaches.[15] The implementation takes the inputs and outputs specified in the ADT, but does not take how these inputs are processed to produce the outputs from the ADT, it implements these itself.[18]

Task1(l):

As mentioned before, an ADT is not an implementation, but only includes the operations, their inputs, and outputs. It does tell how to implement a data structure. In Object-Oriented programming, ADTs are types of objects that that specify values and operations. It does not include how the data will be processed, or how it will be stored in the memory. As its name suggests, it is abstract, as in it is implementation independent.[16]

ADTs are considered a basis for Object-Oriented programming as the ADTs are objects themselves that require implementations to be made out of, for the ADT to be useful. I agree with that view as many different Object-Oriented concepts are shared with ADTs. For example, implementations of an ADT inherit an ADT's values and operations, and can override some operations, depending on the implementation's needs.[17]

Another example is the ADT includes the concepts of encapsulation and abstraction. Encapsulation because an ADT has the details of an implementation hidden from the implementation, and only mentions the basic values and operations. And Abstraction because an ADT is implementation independent, and does not need to be edited to suit a system's requirements, but can instead be built upon by adding operations that the system may require.[17]

Object-Oriented programming language also include the Abstract data type, to allow for easier creation of implementation-independent data structures. This further strengthens the view that imperative ADTs are a basis of Object-Orientation.[18]

Another fact is that when creating a data structure for a system, it would be easier to brainstorm and create an ADT with the basic values, inputs, outputs, and operations of said data structure, before implementing it. This allows for the reusability of this ADT, which means it can be used in many different implementations. Moreover, This provides better abstraction and memory efficiency than creating the implementation without the ADT, as rather than change every implementation when wanting to edit, add, or delete an operation/value from the data structure, we can simply change the ADT and the implementations would follow, which is much more efficient and effective.[18]

Task2:

Task2(a):

There are many algorithms used to find the shortest path from a point to another point. The two explained below are Dijkstra’s algorithm, and Bellman-Ford algorithm.

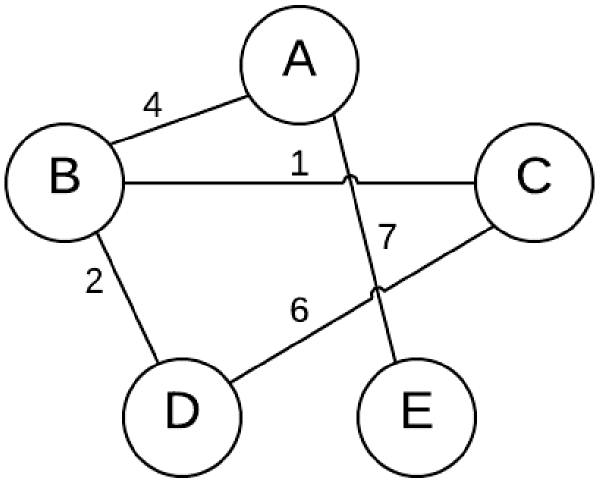
Dijkstra’s algorithm:

This algorithm is done using the Greedy method strategy, which means this algorithm does not calculate the best result from the start, but finds the path with the least distance from the point it is at. This means that does not calculate the overall best path from a point to another directly.[19]

Algorithms that use the Greedy method strategy have high time complexity, because they need to find the best result by building upon smaller results, and not calculating directly.[19]

For example, we have this graph below, and we need to find the shortest path from C to A.

13

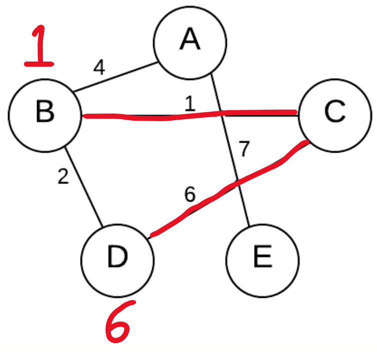


1. First of all, we start with the distance for C itself. All distances from C to the other vertices is considered as unknown distances. In this example we use infinity to represent these unknown distances.

Vertices	Shortest Path to Source Vertex	Previous Vertex
A	∞	
B	∞	
C	0	
D	∞	
E	∞	

2. Then we go to the unvisited vertex with the shortest path, which is currently vertex C.

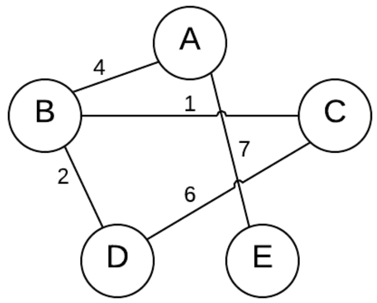
3. Then we calculate the distances from vertex C to the unvisited vertexes that neighbor it, and update the table accordingly.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	∞	
B	1	C
C	0	
D	6	C
E	∞	

`visited = []`

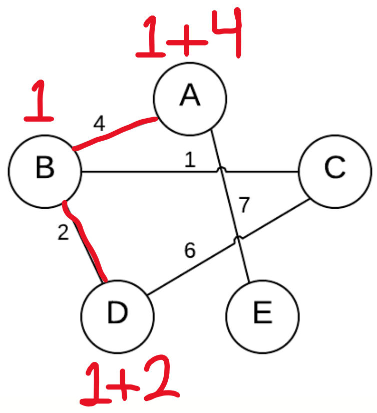
4. After that, C is considered as being visited, and it is added to the visited list.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	∞	
B	1	C
C	0	
D	6	C
E	∞	

`visited = [C]`

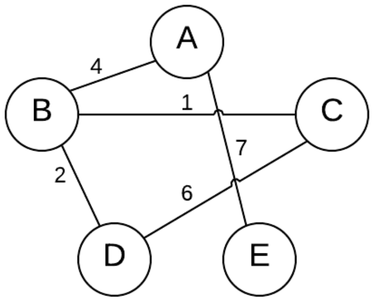
5. Then step 2 and 3 are repeated, where we go to the unvisited vertex with the shortest path, and calculate the distances to the unvisited neighbors to said vertex. We go to vertex B because it has the shortest path. And, because the path to D is shorter than the previous path from C which has a distance of 6, the shortest path to D is updated to the new path, with B as the previous vertex.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	∞	

`visited = [C]`

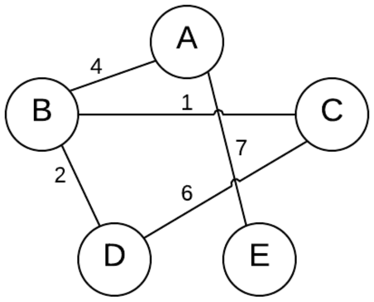
6. Same with step 4, after finishing with B, we add it the visited list, so that we do not repeat this path.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	∞	

visited = [C, B]

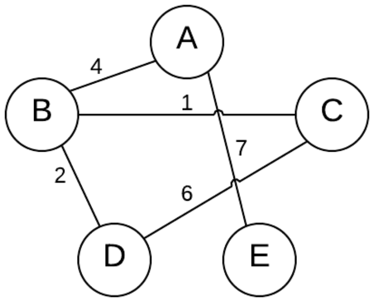
7. Again, we go to the unvisited vertex with the shortest path, which is D, as it has a shorter path than A, and because C and B have already been visited. We calculate the distances to the unvisited neighboring vertices, but as D does not have any unvisited neighbors, nothing changes. This is because B and C have already been visited.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	∞	

visited = [C, B]

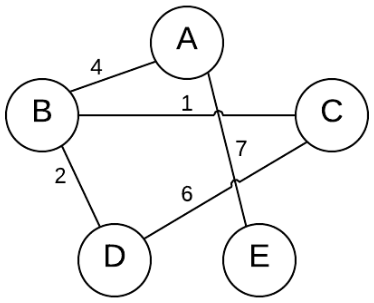
8. Then D is added to the visited list, as we have finished with it.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	∞	

visited = [C, B, D]

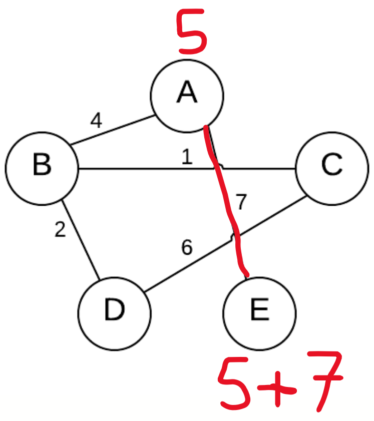
9. We then go the unvisited vertex with the shortest path, which is A, considering that the other vertices have already been visited, and that E still does not have a proper distance to it.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	∞	

`visited = [C, B, D]`

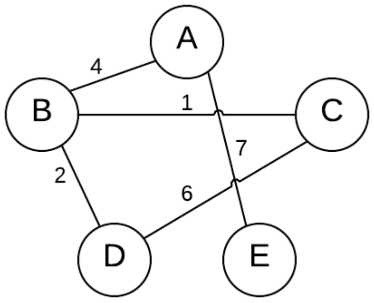
10. Then the distance to the unvisited vertices that neighbor A is calculated, and the table is updated.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	12	A

`visited = [C, B, D]`

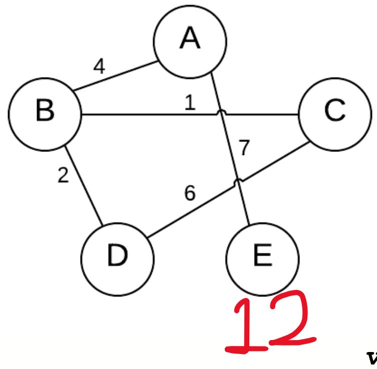
11. After that, A is added to the visited list.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	12	A

`visited = [C, B, D, A]`

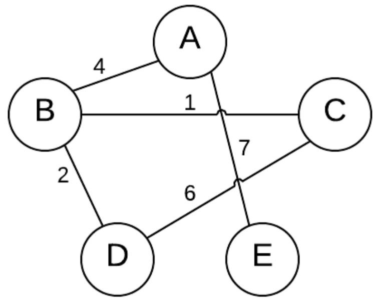
12. Again, we go the unvisited vertex with the shortest path, which is E, considering that all other vertices have been visited.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	12	A

```
visited = [C, B, D, A]
```

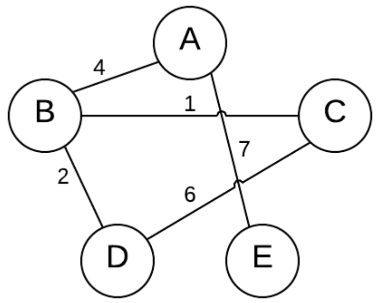
13. E does not have any unvisited neighbors connected to it; therefore, nothing changes to the table.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	12	A

```
visited = [C, B, D, A]
```

14. After that, E is added to the visited list.



Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	12	A

```
visited = [C, B, D, A, E]
```

15. There are no more vertices to visit, therefore, the table is complete, and the algorithm ends.

Vertices	Shortest Path to Source Vertex	Previous Vertex
A	5	B
B	1	C
C	0	
D	3	B
E	12	A

visited = [C, B, D, A, E]

This means that the shortest path from vertex C to vertex A has a distance of 5, and the previous vertex is B. The path would be C → B → A.

Dijkstra's algorithm relaxes the distances of each vertex before each one is visited, and therefore we get the shortest path by having checked all other vertices.[19]

One of the disadvantages of Dijkstra's algorithm is that it does not accept negative weighted edges.[19]

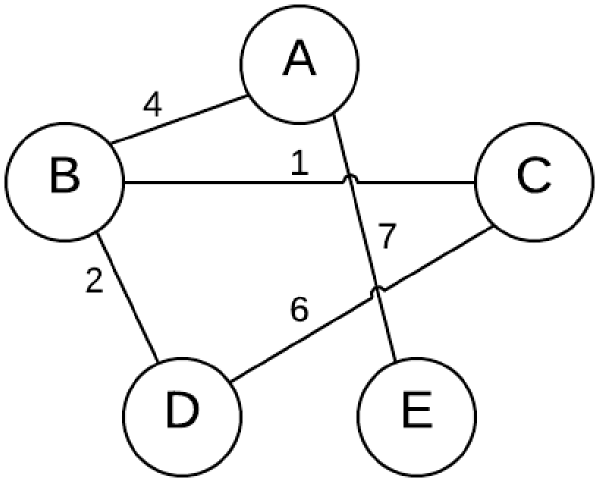
Dijkstra's algorithm has a high time complexity of O(n^2) as it needs to visit all vertices and relax each vertex that neighbors the vertex being visited. By assuming that each vertex is connected to all other vertices, and that the number of vertices is n, this means that we need to loop n * n times, because we first need to loop through all vertices(n), and then loop to relax all vertices for each vertex we visit.[19]

Bellman-Ford algorithm:

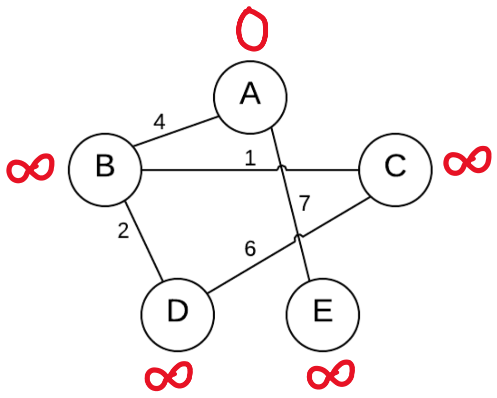
Same with Dijkstra's algorithm, Bellman-Ford algorithm returns the same result; the shortest path from one vertex to all other vertices. This algorithm does not use the Greedy method, but still has a high time complexity. Moreover, it works on graphs that have negative weighted edges. It works by relaxing all edges of the graph an (n – 1) number of times, where n is the number of vertices. Additionally, this algorithm does not specify the previous vertex of each path, only the distance.[19]

For example, we need the shortest path from vertex A, to all other vertices in the graph below:

13

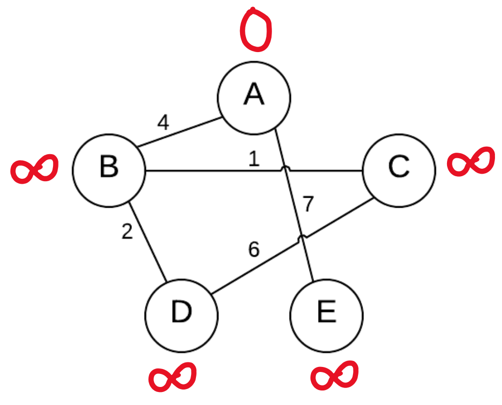


1. For this algorithm to work, first we need a list of the edges of the graph, and initialize the distances of all vertices as unknown distances(infinity value chosen), except for the source vertex, which has its distance set to 0.



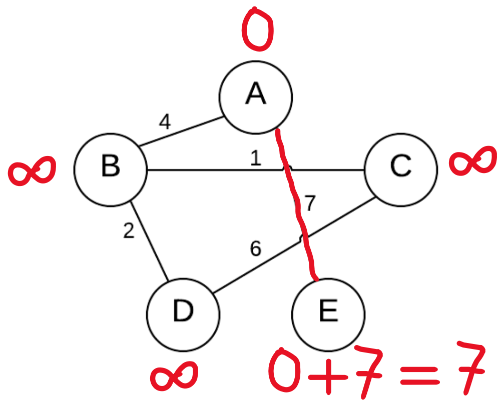
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

2. The next step is to relax all edges. There are 5 vertices, therefore, the edges should be relaxed 4 times.



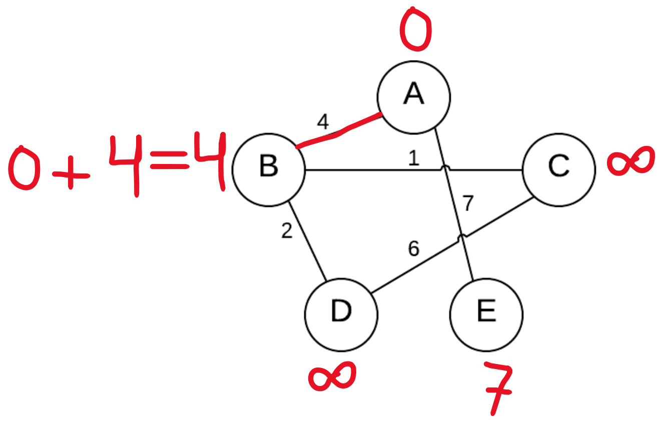
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

3. We then go to the first edge, which is (E, A), and relax its weight.

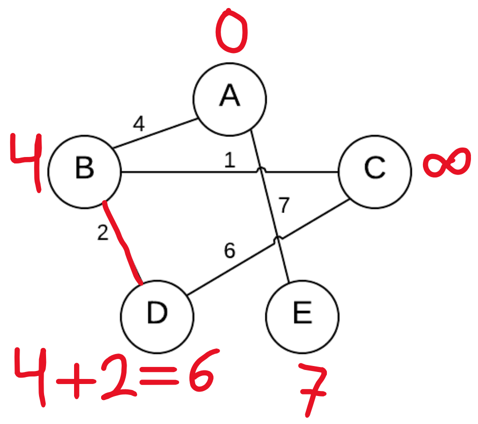


Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

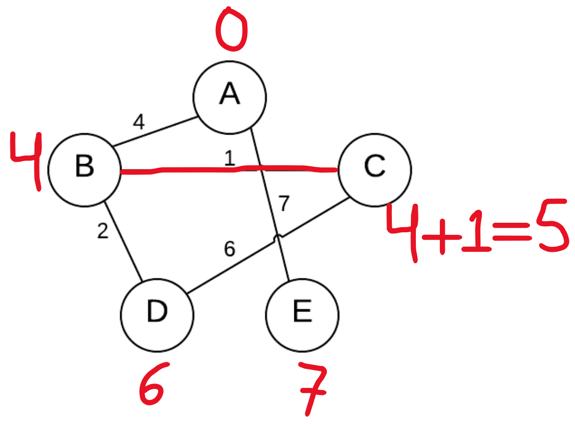
4. Repeat the process of going to the next edge, and relaxing their weights.



Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

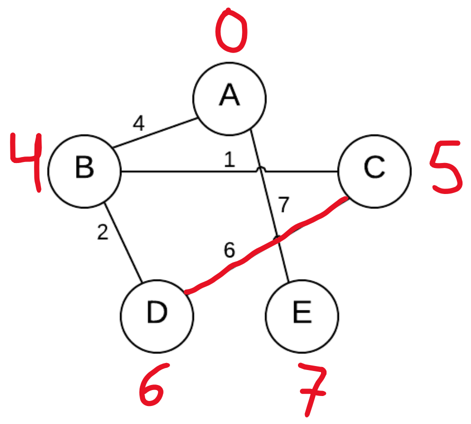


Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)



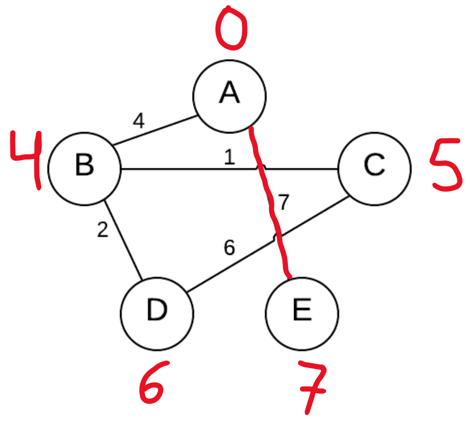
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

5. When relaxing an edge and giving a new distance to the vertex, we need to check which one has the less weight among the two distances, and then update the distance to the shorter one. In the graph, C vertex already has the distance 5, but the edge (D, C) may overwrite this distance. The edge (D, C) gives C the distance of 11, which is more than 5, therefore, the distance is not updated.

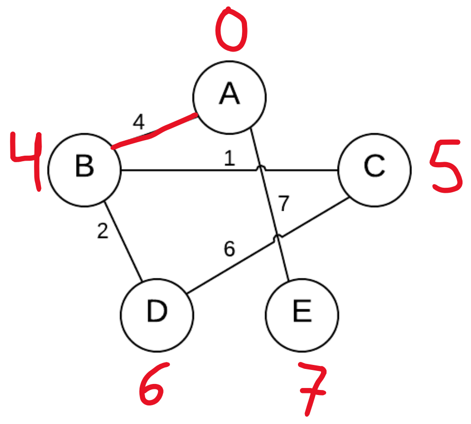


Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

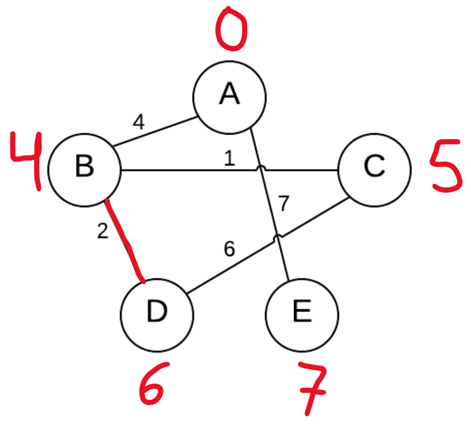
6. We have finished the 1st iteration of all edges. Now onto the 2nd iteration. Because the path from A to E has the same distance, it is not relaxed.



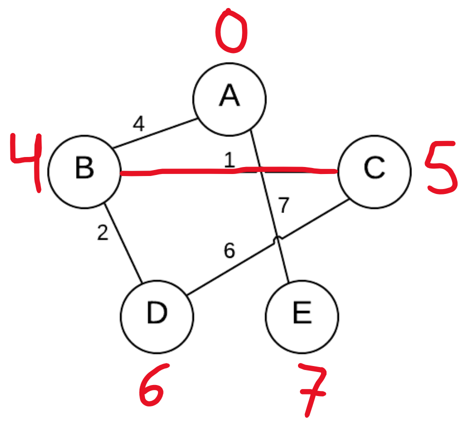
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)



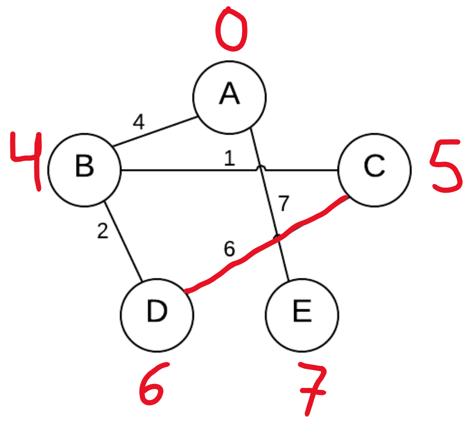
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)



Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

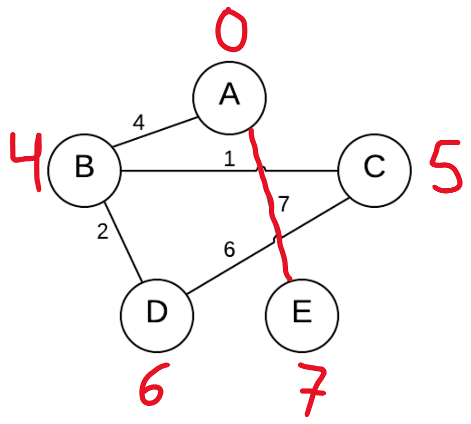


Edges
(E, A)
(A, B)
(B, D)
🟥 (B, C)
(D, C)

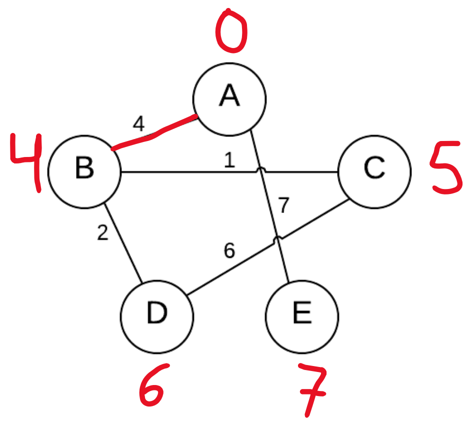


Edges
(E, A)
(A, B)
(B, D)
(B, C)
🟥 (D, C)

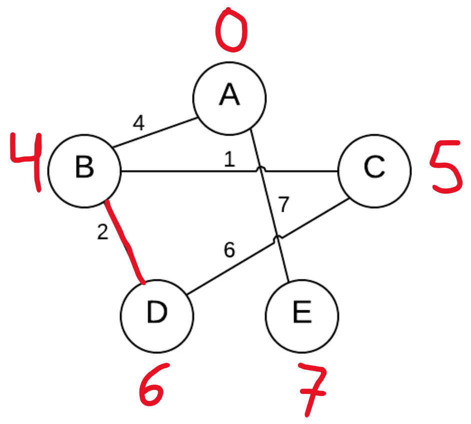
7. We have finished the 2nd iteration of all edges. Now onto the 3rd iteration. The same processes, as in step 6, are repeated.



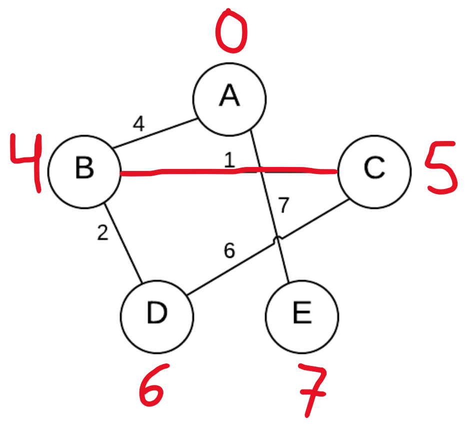
Edges
🟥 (E, A)
(A, B)
(B, D)
(B, C)
(D, C)



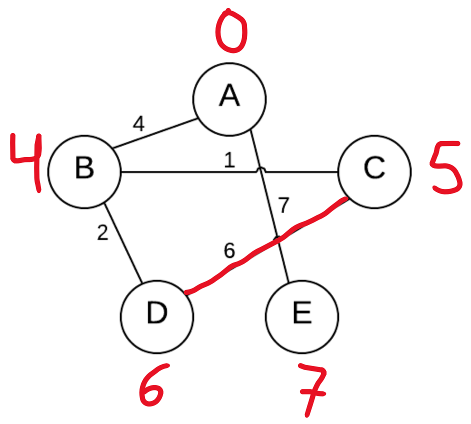
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)



Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

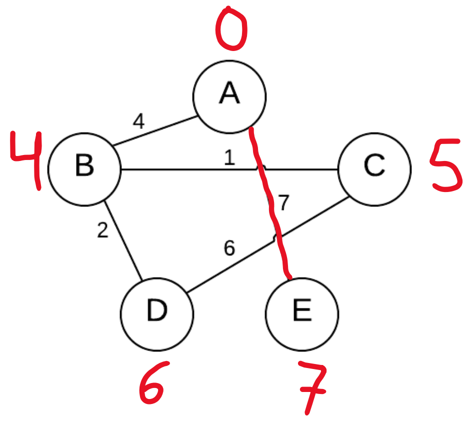


Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

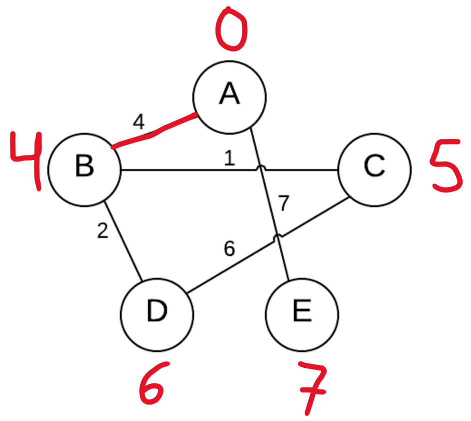


Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

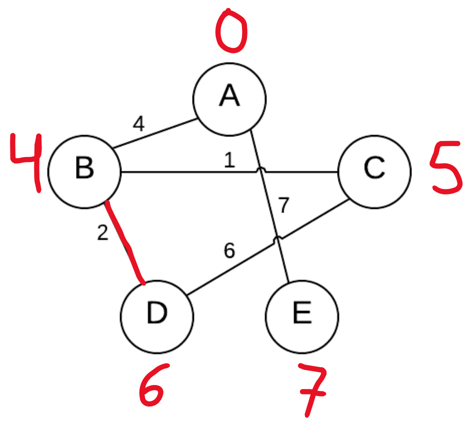
8. We have finished the 3rd iteration of all edges. Now onto the 4th and last iteration. The same processes, as in steps 6 and 7, are repeated.



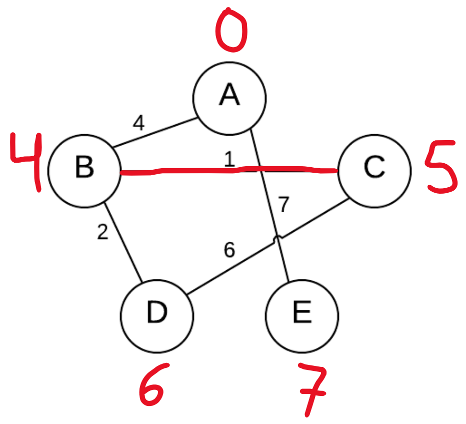
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)



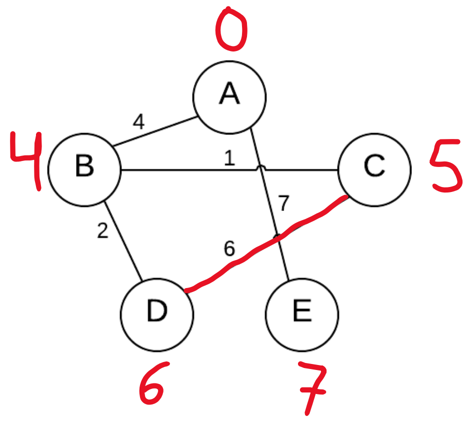
Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)



Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

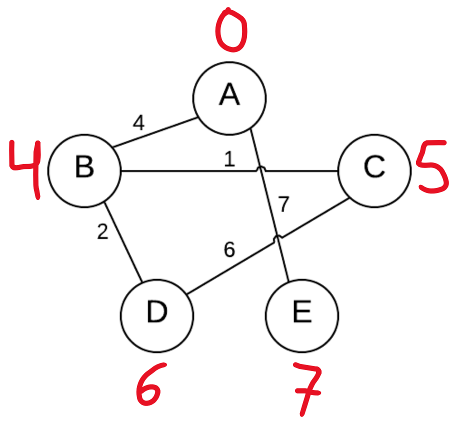


Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)



Edges
(E, A)
(A, B)
(B, D)
(B, C)
(D, C)

9. After going through (n – 1) iterations, where n is the number of vertices, all edges have relaxed, and now we have the shortest distances from vertex A to all other vertices; thus, the algorithm ends.



This means that to travel from vertex A to any other vertex, the shortest paths would have distances as follows:

- A to B has a distance of 4.
- A to C has a distance of 5.
- A to D has a distance of 6.
- A to E has a distance of 7.

One of disadvantages of Bellman-Ford algorithm is that it does not calculate the paths, as Dijkstra's algorithm does, but rather only calculates the distance. This means that we do not know where and how to go from vertex A to C, but only know that the shortest distance from A to C is 5.[18]

An advantage that Bellman-Ford algorithm provides is the fact that it can accept negative weighted edges, without problems occurring with the result. Moreover, it does not rely on the Greedy method strategy.[19]

In the example above, an operation could be added to end the algorithm if no edges were relaxed in an iteration. This could lessen time complexity, but the worst-case scenario would remain having O($e * n$), where n is the number of vertices, and e is the number of edges. This is because we need to go through all edges(e) an n – 1 number of times. This makes it O($e * (n - 1)$), and because it is asymptotic analysis, it becomes O($e * n$).[18]

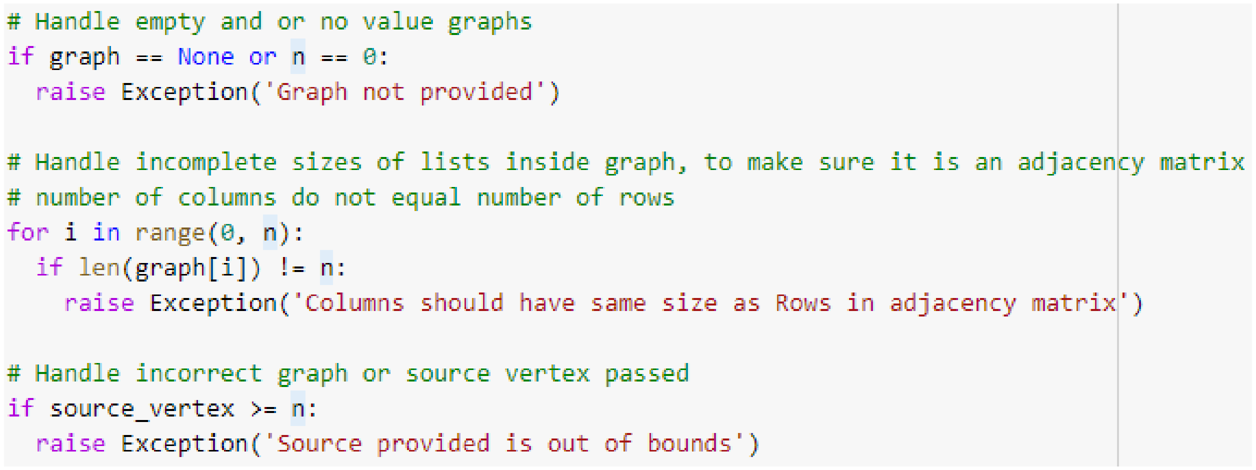
This algorithm works by considering all edges an (n – 1), where n is the number of vertices, number of times, one by one, and then relaxing the sum of their weights from source to vertices of edges.[18]

Task2(b):

Code is provided in ***task2(b).py*** submitted file.

Task2(c):

Error handling was implemented. This part of the code represents it:



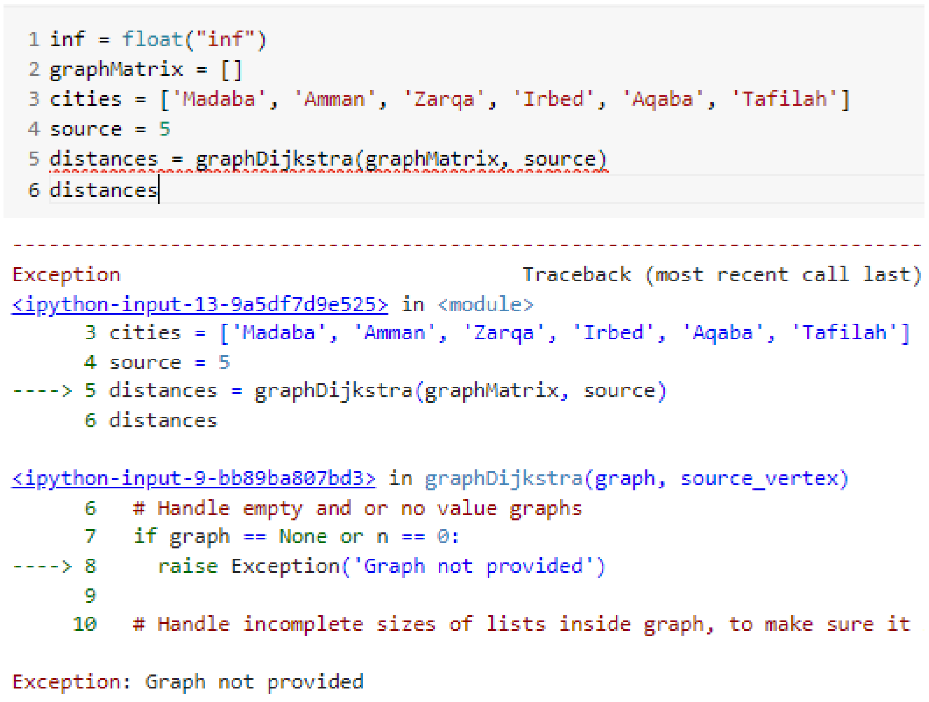
```python
# Handle empty and or no value graphs
if graph == None or n == 0:
  raise Exception('Graph not provided')

# Handle incomplete sizes of lists inside graph, to make sure it is an adjacency matrix
# number of columns do not equal number of rows
for i in range(0, n):
  if len(graph[i]) != n:
    raise Exception('Columns should have same size as Rows in adjacency matrix')

# Handle incorrect graph or source vertex passed
if source_vertex >= n:
  raise Exception('Source provided is out of bounds')
```

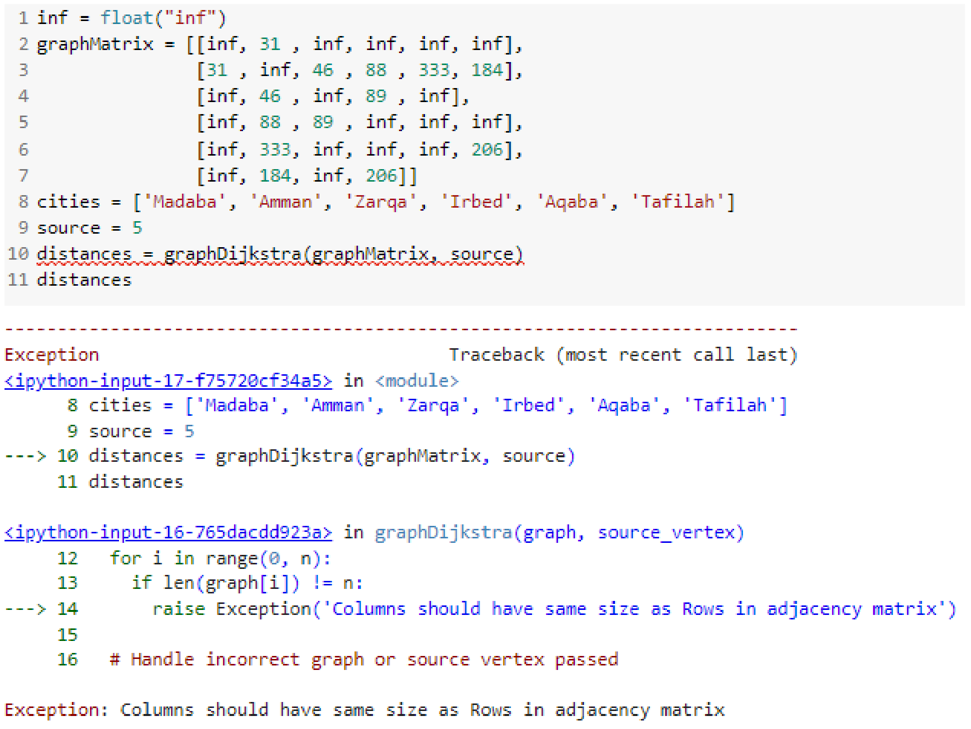
In the first part, it makes sure that the graph is not empty, and is a list. Otherwise, the algorithm will not work.

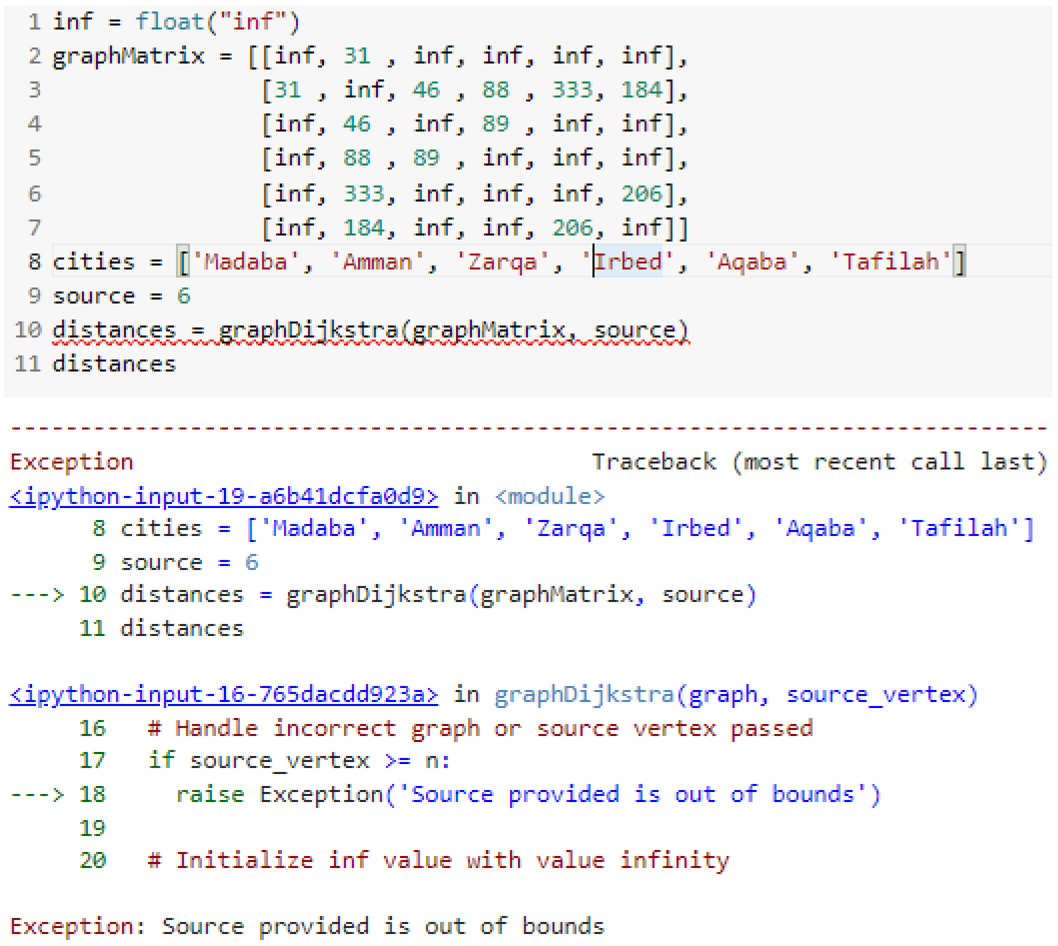
In the second part, it makes sure that the graph provided is in the format of an adjacency matrix, with its columns and rows equal in size. Otherwise, it will cause complications with the algorithm and may cause to not work properly.

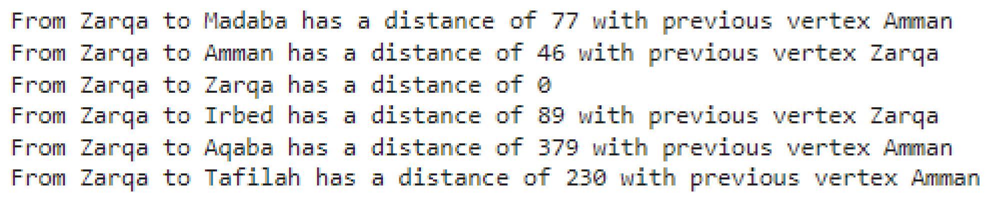
In the third part, it makes sure that the source vertex provided is a valid vertex number, not equal nor more than the number of rows in the list.

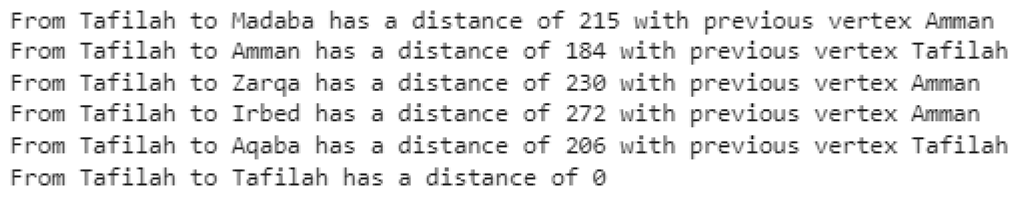
Tests:

Test	Results	Description
Empty list as Graph	 inf = float("inf") graphMatrix = [] cities = ['Madaba', 'Amman', 'Zarqa', 'Irbed', 'Aqaba', 'Tafilah'] source = 5 distances = graphDijkstra(graphMatrix, source) distances --- Exception Traceback (most recent call last) <ipython-input-13-9a5df7d9e525> in <module> 3 cities = ['Madaba', 'Amman', 'Zarqa', 'Irbed', 'Aqaba', 'Tafilah'] 4 source = 5 ----> 5 distances = graphDijkstra(graphMatrix, source) 6 distances <ipython-input-9-bb89ba807bd3> in graphDijkstra(graph, source_vertex) 6 # Handle empty and or no value graphs 7 if graph == None or n == 0: ----> 8 raise Exception('Graph not provided') 9 10 # Handle incomplete sizes of lists inside graph, to make sure it Exception: Graph not provided	An empty list was given to the Dijkstra's algorithm function, and an exception was raised because the function handled an empty graph as an error.

Number of Columns in matrix do not equal number of Rows	
	inf = float("inf")
	graphMatrix = [[inf, 31 , inf, inf, inf, inf],
	               [31 , inf, 46 , 88 , 333, 184],
	               [inf, 46 , inf, 89 , inf],
	               [inf, 88 , 89 , inf, inf, inf],
	               [inf, 333, inf, inf, inf, 206],
	               [inf, 184, inf, 206]]
	cities = ['Madaba', 'Amman', 'Zarqa', 'Irbed', 'Aqaba', 'Tafilah']
	source = 5
	distances = graphDijkstra(graphMatrix, source)
	distances
	
	--- 	
	Exception Traceback (most recent call last)
	<ipython-input-17-f75720cf34a5> in <module>
	      8 cities = ['Madaba', 'Amman', 'Zarqa', 'Irbed', 'Aqaba', 'Tafilah']
	      9 source = 5
	---> 10 distances = graphDijkstra(graphMatrix, source)
	     11 distances
	
	<ipython-input-16-765dacdd923a> in graphDijkstra(graph, source_vertex)
	     12 for i in range(0, n):
	     13 if len(graph[i]) != n:
	---> 14 raise Exception('Columns should have same size as Rows in adjacency matrix')
	     15
	     16 # Handle incorrect graph or source vertex passed
	
	Exception: Columns should have same size as Rows in adjacency matrix
		A matrix that does not have the same number of columns as number of rows was passed to Dijkstra's algorithm, and an exception occurred because function handled this as an error.

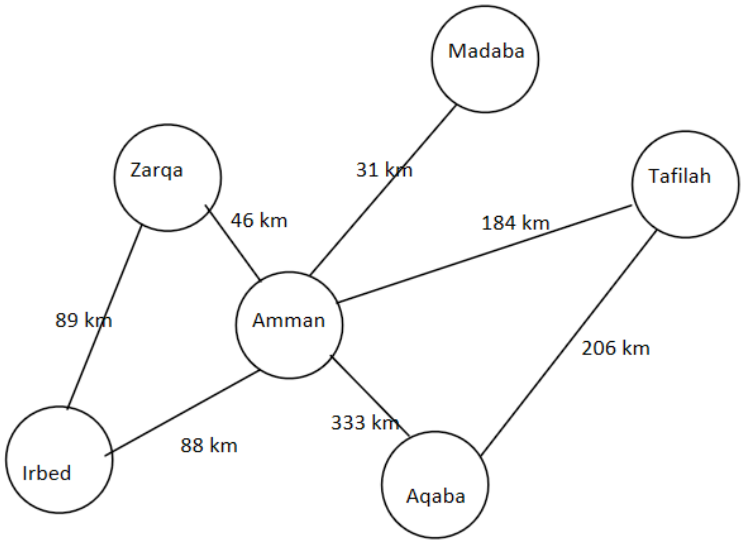
Source larger than number of Rows	
	`inf = float("inf")`
	`graphMatrix = [[inf, 31 , inf, inf, inf, inf],`
	`               [31 , inf, 46 , 88 , 333, 184],`
	`               [inf, 46 , inf, 89 , inf, inf],`
	`               [inf, 88 , 89 , inf, inf, inf],`
	`               [inf, 333, inf, inf, inf, 206],`
	`               [inf, 184, inf, inf, 206, inf]]`
	`cities = ['Madaba', 'Amman', 'Zarqa', 'Irbed', 'Aqaba', 'Tafilah']`
	`source = 6`
	`distances = graphDijkstra(graphMatrix, source)`
	`distances`
	
	`---------------------------------------------------------------------------`
	`Exception                                 Traceback (most recent call last)`
	`<ipython-input-19-a6b41dcfa0d9> in <module>`
	`      8 cities = ['Madaba', 'Amman', 'Zarqa', 'Irbed', 'Aqaba', 'Tafilah']`
	`      9 source = 6`
	`---> 10 distances = graphDijkstra(graphMatrix, source)`
	`     11 distances`
	
	`<ipython-input-16-765dacdd923a> in graphDijkstra(graph, source_vertex)`
	`     16     # Handle incorrect graph or source vertex passed`
	`     17     if source_vertex >= n:`
	`---> 18       raise Exception('Source provided is out of bounds')`
	`     19`
	`     20     # Initialize inf value with value infinity`
	
	`Exception: Source provided is out of bounds`
		A source vertex index was given to Dijkstra's algorithm function, and an exception occurred because the function handles a source vertex index bigger than the number of rows in the graph as an error.

Source vertex set to Zarqa	
	`From Zarqa to Madaba has a distance of 77 with previous vertex Amman`
	`From Zarqa to Amman has a distance of 46 with previous vertex Zarqa`
	`From Zarqa to Zarqa has a distance of 0`
	`From Zarqa to Irbed has a distance of 89 with previous vertex Zarqa`
	`From Zarqa to Aqaba has a distance of 379 with previous vertex Amman`
	`From Zarqa to Tafilah has a distance of 230 with previous vertex Amman`
		The shortest path from Zarqa to all other cities is shown.

Source vertex set to Tafilah	
	`From Tafilah to Madaba has a distance of 215 with previous vertex Amman`
	`From Tafilah to Amman has a distance of 184 with previous vertex Tafilah`
	`From Tafilah to Zarqa has a distance of 230 with previous vertex Amman`
	`From Tafilah to Irbed has a distance of 272 with previous vertex Amman`
	`From Tafilah to Aqaba has a distance of 206 with previous vertex Tafilah`
	`From Tafilah to Tafilah has a distance of 0`
		The shortest path from Tafilah to all other cities is shown.

Task2(d):

To find the shortest path from the two cities, Zarqa and Tafilah, we need a shortest path algorithm. We are going to use the Dijkstra algorithm. This algorithm uses a graph data structure, which can have many different ways if representation, but the algorithm used uses an adjacency matrix. The following graph is implemented as an adjacency matrix:



```
graphMatrix = [[inf, 31 , inf, inf, inf, inf],
               [31 , inf, 46 , 88 , 333, 184],
               [inf, 46 , inf, 89 , inf, inf],
               [inf, 88 , 89 , inf, inf, inf],
               [inf, 333, inf, inf, inf, 206],
               [inf, 184, inf, inf, 206, inf]]
```

My implementation uses the adjacency matrix as the graph, and finds the shortest path from Zarqa to Tafilah. It follows the following steps:

1. First of all, the algorithm's function is called with the graph and the source vertex index, which is 2 for Zarqa, passed.

   ```
   distances = graphDijkstra(graphMatrix, source)
   ```

2. In the function, the size of the graph is calculated because it represents the number of vertices.

```
# Function that takes a graph and source, uses Dijkstra algorithm to find shortest path from source to all other vertices
def graphDijkstra(graph, source_vertex):
  # Get length of graph, which is number of vertices
  n = len(graph)
```

3. The next processes are error handling, and validating the graph and source vertex provided.

```python
# Handle empty and or no value graphs
if graph == None or n == 0:
  raise Exception('Graph not provided')

# Handle incomplete sizes of lists inside graph, to make sure it is an adjacency matrix
# number of columns do not equal number of rows
for i in range(0, n):
  if len(graph[i]) != n:
    raise Exception('Columns should have same size as Rows in adjacency matrix')

# Handle incorrect graph or source vertex passed
if source_vertex >= n:
  raise Exception('Source provided is out of bounds')
```

   The description of these error handlers is in Task2(c).

4. Then the shortest distances from the source to all vertices are initialized with any value that is considered an invalid distance, where in our case, the infinity value is used, and the distance from the source vertex to itself is set to 0.

```python
# Initialize inf value with value infinity
inf = float("inf")

# Intialize distances of vertices with infinity value, and set source vertex distance to itself to 0
distances = [inf] * n
distances[source_vertex] = 0
```

5. Then two lists are initialized, one to save the visited vertices so that they are not repeated, and the other is to save the previous vertex of each vertex path. Both have the same size as the number of vertices.

```python
# visited list to save already visited vertices, to not repeat them
visited = [None] * n
# previous list to store the previous vertex of each path to each vertex.
previous = [None] * n
```

6. After that, the core processes of this algorithm are started. We go through each vertex, calculate the shortest distance to neighboring vertices, go to them, and the same process is repeated again, until all vertices are checked.

```python
# loop to iterate through all vertices
for i in range(0, n):
  # Intialize next variable with value -1 to save the vertex with the shortest path from current vertex, which is i
  next = -1
  # iterate through all vertices to find the shortest connected path
  for j in range(0, n):
    # check if vertex has already been visited, or if the distance to vertex j is smaller than saved distance
    if (distances[j] < distances[next] or next == -1) and visited[j] == None:
      # set next vertex as vertex j with shortest path
      next = j

  # add next vertex to visited to repeat it
  visited[next] = next
  # iterate through all vertices to relax distances, and save previous vertex
  for j in range(0, n):
    # make sure that there is a path between vertices, and that overall distance is less
    # than distance saved
    if graph[next][j] != inf and (distances[next] + graph[next][j]) < distances[j]:
      # update to new distance
      distances[j] = distances[next] + graph[next][j]
      previous[j] = next # save previous vertex for paths of all vertices
```

7. Lastly, the function returns the final shortest distances from source vertex, which is Zarqa, to all other vertices, including Tafilah. The previous list is returned along with the distances, and the same indices of both refer to the same vertices.

```python
# return distances and previous vertices of these distances
return [distances, previous]
```

The shortest paths from source vertex, Zarqa, to all vertices, including the target, Tafilah, are returned, with their previous vertices included:

```
[[77, 46, 0, 89, 379, 230], [1, 2, None, 2, 1, 1]]
```

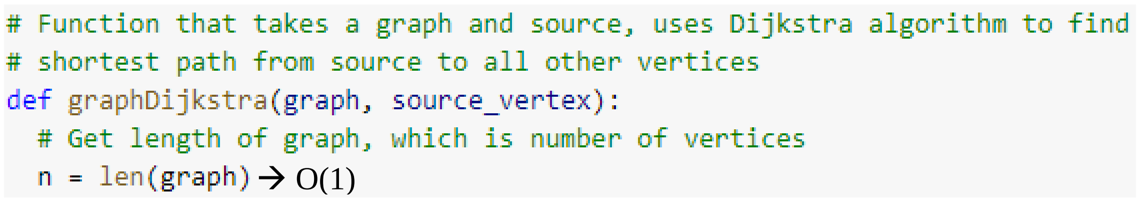
After formatting the data as in both lists, the same indices refer to the same vertex, where it is from the source vertex to all vertices, including the distance with the previous vertex of each path:

```
From Zarqa to Madaba has a distance of 77 with previous vertex Amman
From Zarqa to Amman has a distance of 46 with previous vertex Zarqa
From Zarqa to Zarqa has a distance of 0
From Zarqa to Irbed has a distance of 89 with previous vertex Zarqa
From Zarqa to Aqaba has a distance of 379 with previous vertex Amman
From Zarqa to Tafilah has a distance of 230 with previous vertex Amman
```

The last line is the one that provides that shortest path from Zarqa to Tafilah.

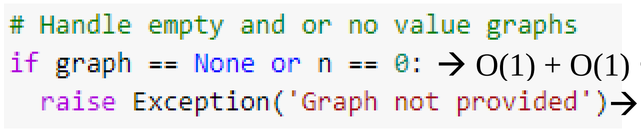
And considering the function above, it is possible to use many different graphs, just that the function will only accept graphs in the adjacency matrix format.

Task2(e):



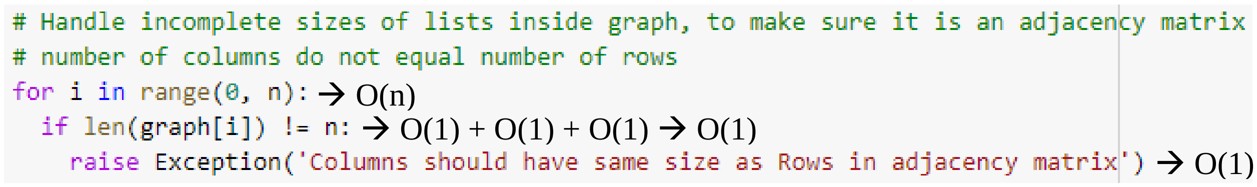
```
# Function that takes a graph and source, uses Dijkstra algorithm to find
# shortest path from source to all other vertices
def graphDijkstra(graph, source_vertex):
  # Get length of graph, which is number of vertices
  n = len(graph) → O(1)
```

The code above has an overall time complexity of O(1) because the len() function in python returns the size of the graph from a size variable stored in the list, not by looping through the elements of the graph and counting them.



```
# Handle empty and or no value graphs
if graph == None or n == 0: → O(1) + O(1) → O(1)
  raise Exception('Graph not provided')→ O(1)
```

The code above has an overall time complexity of O(1) because it is only doing simple operations using the equality operator. As it is asymptotic, from O(1) + O(1) it becomes O(1).



```
# Handle incomplete sizes of lists inside graph, to make sure it is an adjacency matrix
# number of columns do not equal number of rows
for i in range(0, n): → O(n)
  if len(graph[i]) != n: → O(1) + O(1) + O(1) → O(1)
    raise Exception('Columns should have same size as Rows in adjacency matrix') → O(1)
```

The code above has an overall time complexity of O(n) because it needs to loop n times through code that has a time complexity of O(1).

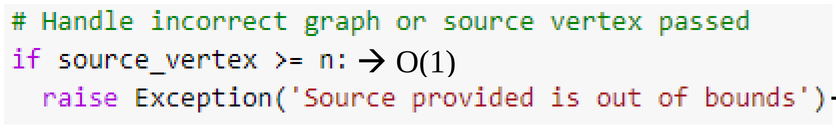
The (graph[i]) has a time complexity of O(1) because it is simply calling a value in a certain index.

The len() function has a time complexity of O(1).

The time complexity of the not equal operator is also O(1).

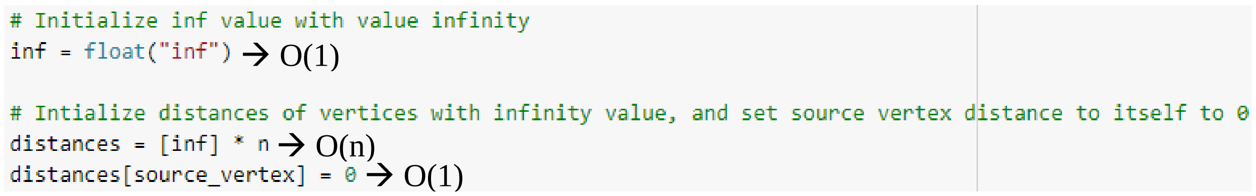
As it is asymptotic, from O(1) + O(1) + O(1) it becomes O(1).

n * O(1) = O(n)



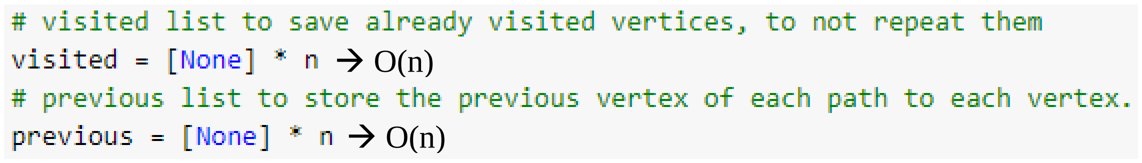
```
# Handle incorrect graph or source vertex passed
if source_vertex >= n: → O(1)
  raise Exception('Source provided is out of bounds')→ O(1)
```

This has an overall time complexity of O(1) because it is only doing simple operations using the greater than or equal operator.



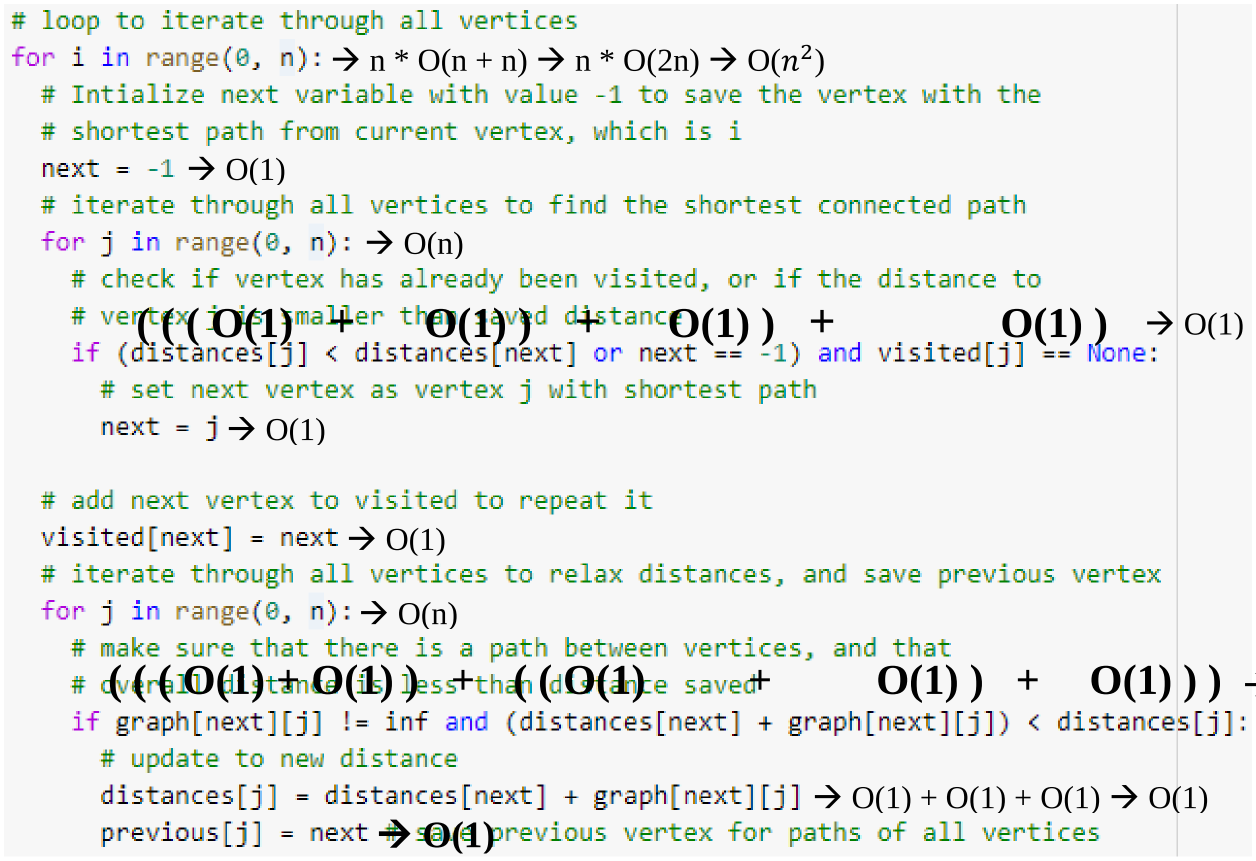
```
# Initialize inf value with value infinity
inf = float("inf") → O(1)

# Intialize distances of vertices with infinity value, and set source vertex distance to itself to 0
distances = [inf] * n → O(n)
distances[source_vertex] = 0 → O(1)
```

This has an overall time complexity of O(n) because it is initializing a list that has a variable size of n, where n is the number of vertices. The other lines only have a time complexity of O(1) because they are simple assignment operations. As is asymptotic, O(1) + O(n) + O(1) becomes O(n), because we use the worst-case scenario.



```
# visited list to save already visited vertices, to not repeat them
visited = [None] * n → O(n)
# previous list to store the previous vertex of each path to each vertex.
previous = [None] * n → O(n)
```

This has an overall time complexity of O(n) because both operations are initializing a list that has a variable size of n, where n is the number of vertices. As it is asymptotic, O(n) + O(n) becomes O(n).



```
# loop to iterate through all vertices
for i in range(0, n): → n * O(n + n) → n * O(2n) → O(n²)
  # Intialize next variable with value -1 to save the vertex with the
  # shortest path from current vertex, which is i
  next = -1 → O(1)
  # iterate through all vertices to find the shortest connected path
  for j in range(0, n): → O(n)
    # check if vertex has already been visited, or if the distance to
    # vertex j is smaller than saved distance
       (( ( O(1) + O(1) ) + O(1) ) + O(1) ) → O(1)
    if (distances[j] < distances[next] or next == -1) and visited[j] == None:
      # set next vertex as vertex j with shortest path
      next = j → O(1)

  # add next vertex to visited to repeat it
  visited[next] = next → O(1)
  # iterate through all vertices to relax distances, and save previous vertex
  for j in range(0, n): → O(n)
    # make sure that there is a path between vertices, and that
    # overall distance is less than distance saved
       ( ( ( O(1) + O(1) ) + ( ( O(1) + O(1) ) + O(1) ) ) → O(1)
    if graph[next][j] != inf and (distances[next] + graph[next][j]) < distances[j]:
      # update to new distance
      distances[j] = distances[next] + graph[next][j] → O(1) + O(1) + O(1) → O(1)
      previous[j] = next → O(1) # save previous vertex for paths of all vertices
```

This has an overall time complexity of O(n^2) because it loops n times through code that a time complexity of O(n). By adding the time complexity of each small part together asymptoticly, we can get the overall time complexity of this part of the algorithm.

(`next = -1`) and (`next = j`) each have a time complexity of O(1) as they are only simple assignment operations. Asymptoticly, their overall time complexity is O(1) as well.

(`distances[j]`), (`distances[next]`), (`visited[j]`), (`visited[next]`), (`graph[next][j]`), and (`previous[j]`) each have a time complexity of O(1) as they are only calls for values from specific indices. Asymptoticly, their overall time complexity is O(1).

(`visited[next] = next`), (`distances[j] = distances[next] + graph[next][j]`), and (`previous[j] = next`) each have a time complexity of O(1) as they are simply assigning values to specific indices. Asymptoticly, their overall time complexity is O(1).

(`next == -1`), (`visited[j] == None`), and (`graph[next][j] != inf`) each have a time complexity of O(1) as they are simple conditional operators, specifically the equality and not equal operators. Asymptoticly, their overall time complexity is O(1).

(`distances[j] < distances[next] or next == -1`),
(`(distances[j] < distances[next] or next == -1) and visited[j] == None`),
(`distances[next] + graph[next][j]`),
(`(distances[next] + graph[next][j]) < distances[j]`), and
(`graph[next][j] != inf and (distances[next] + graph[next][j]) < distances[j]`)
each also have a time complexity of O(1) as they are logical and comparison operators. Asymptotically, their overall time complexity is O(1).

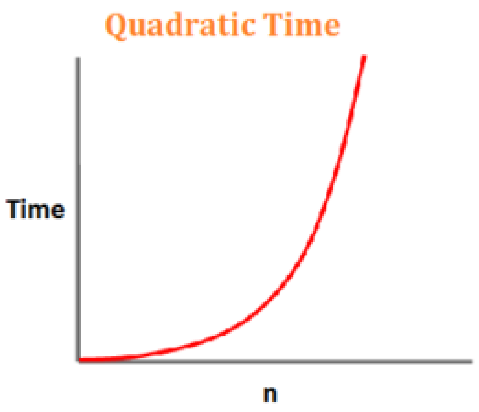
(`for j in range(0, n):`), and (`for j in range(0, n):`) each have a time complexity of O(n) as they are looping n times for code that, asymptotically, has a time complexity of O(1).

n * O(1) = O(n).

Together, their time complexity is calculated as such: O(n + n) → O(2n) → O(n)

(`for i in range(0, n):`) on its own has a time complexity of O(n), but considering that the overall time complexity inside this loop is O(n), it is calculated as such: n * O(n) → O(n^2). This means that this part of the algorithm has a Quadratic runtime.

14



```
# return distances and previous vertices of these distances
return [distances, previous]
```
→ O(n)

This final part of the algorithm that returns the distances and the previous vertices for each path has a time complexity of O(n) because the sizes of the distances and previous lists are n, and that is a variable that represents the number of vertices in the graph.

Adding together the time complexity of the different parts of the algorithm:

Asymptotically, O$(1 + 1 + n + 1 + n + n + n^2 + n)$ becomes O(n^2) as the bigO notation takes the worst-case scenario, which in our case is the time complexity being O(n^2).

References:

1. Tutorialspoint.com. n.d. *Data Structures - Asymptotic Analysis*. [online] Available at: <https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm> [Accessed 15 August 2022].
2. Kumari, M., 2022. *Efficiency of an Algorithm*. [online] Byjusexamprep.com. Available at: <https://byjusexamprep.com/efficiency-of-an-algorithm-i> [Accessed 16 August 2022].
3. Citizenchoice.in. n.d. *CitizenChoice*. [online] Available at: <https://citizenchoice.in/course/Data-Structures-and-Algorithms/Chapter%201/Time-Space-Trade-off-Abstract-Data-Types-ADT-> [Accessed 16 August 2022].
4. GeeksforGeeks. 2022. *Time-Space Trade-Off in Algorithms - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/time-space-trade-off-in-algorithms/> [Accessed 16 August 2022].
5. Assignmentexpert.com. 2020. *Answer in Java | JSP | JSF for Rakan #149033*. [online] Available at: <https://www.assignmentexpert.com/homework-answers/programming-and-computer-science/java-jsp-jsf/question-149033> [Accessed 20 August 2022].
6. Rehman, H., 2021. *Data Structure and Its Advantages*. [online] Medium. Available at: <https://medium.com/@haseeb26.07/data-structure-and-its-advantages-3e9c2514c9c8> [Accessed 20 August 2022].
7. Burleson, D., n.d. *Encapsulation and Abstract Data Types (ADT)*. [online] Dba-oracle.com. Available at: <http://www.dba-oracle.com/t_object_encapsulation_abstract.htm> [Accessed 20 August 2022].
8. Bolton, D., 2019. *What Is Encapsulation in C++ and C#?*. [online] ThoughtCo. Available at: <https://www.thoughtco.com/definition-of-encapsulation-958068> [Accessed 20 August 2022].
9. Groups.google.com. 2002. *Benefits of Encapsulation*. [online] Available at: <https://groups.google.com/g/comp.object/c/BgTZhfNYazQ?pli=1> [Accessed 20 August 2022].
10. OpenGenus IQ: Computing Expertise & Legacy. n.d. *Time & Space Complexity of Selection Sort*. [online] Available at: <https://iq.opengenus.org/time-complexity-of-selection-sort/> [Accessed 23 August 2022].
11. GeeksforGeeks. 2022. *Merge Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/merge-sort/> [Accessed 23 August 2022].
12. 2022. *Merge Sort*. [image] Available at: <https://www.geeksforgeeks.org/merge-sort-vs-insertion-sort/> [Accessed 24 August 2022].
13. Chumbley, A., n.d. *Shortest Path Algorithms | Brilliant Math & Science Wiki*. [online] Brilliant.org. Available at: <https://brilliant.org/wiki/shortest-path-algorithms/> [Accessed 2 September 2022].
14. Lawton, L., 2017. *Quadratic time Algorithms O(n2)*. [online] Daimto. Available at: <https://www.daimto.com/algorithms-quadratic-time/> [Accessed 2 September 2022].
15. Burleson, D., n.d. *Encapsulation and Abstract Data Types (ADT)*. [online] Dba-oracle.com. Available at: <http://www.dba-oracle.com/t_object_encapsulation_abstract.htm> [Accessed 2 September 2022].
16. GeeksforGeeks. 2022. *Abstract Data Types - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/abstract-data-types/> [Accessed 2 September 2022].

17. Singh, H., 2020. *Object Oriented Programming in Python | OOPs Concepts Python*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2020/09/object-oriented-programming/> [Accessed 3 September 2022].
18. Shamlawi, M. (2022). Al-Hussein Technical University.
19. Lecture Notes.