Starts with calling the user to input a number, to set the range of the random list to be generated

n\_number = int(input("Insert list of numbers to be generated : ")) ## variable for length of list

Create a emptylist called numberlist

number\_list = []F

We use that value, and pass it into a function called createlist ,

createlist(n\_number)

Create List function

Runs a for loop, which appends random integers to the empty numberlist using randint . i set a range for 0,1000, this is an arbitrary choice

List length range is determined by user input

Returns output of a populated numberlist

def createlist(n\_number):

for i in range(0,n\_number): ## range / length of the list from 0 to user input

number\_list.append(random.randint(0,1000)) # Random int appended to the list

return number\_list

We print the new random number\_list for the requirement of assignment 6.b showing list before sort

# Print the list before sorting

print("Before sorting:", number\_list)

b. show the list before and after the sort. -

Next we call the quicksort function,

sorted\_list = quicksort(number\_list)

first checks number\_list <= 1 [Explained later]

if len(number\_list) <= 1:

return number\_list

Assuming the user created a list larger than 1 we choose a pivot value from the newly created random number list

We do this in a new function named choose pivot

# Choose a pivot

pivot = choose\_pivot(number\_list)

Choose pivot function

The aim is to take three numbers , one from the start of the list, one from the end, and one from the middle,

first is called by calling index 0

middle value, by index value = length of numberlist //2 ( // returns full number)  
( note this isn’t perfect, but close enough depending on odd or even number list length, we get to within 1 space of the correct best value )

last index -1,

def choose\_pivot(number\_list):

first = number\_list[0]

middle = number\_list[len(number\_list) // 2]

last = number\_list[-1]

we sort them manually using an if , elif, else statement using greater than less than logic, to find the median

# Find the median of first, middle, and last without sorting

if (first <= middle <= last) or (last <= middle <= first):

pivot = middle

elif (middle <= first <= last) or (last <= first <= middle):

pivot = first

else:

pivot = last

return pivot

The hope being that these are somewhat close to a true median, but this is a random factor in this sorting algorithm

Performance based on pivot chosen:

The best case is when the pivot splits the list into two roughly equal halves, leading to optimal performance

Having two halves is good because it allows the algorithm to recursively sort smaller and smaller sublists, with each step reducing the problem size by approximately half. This balanced partitioning ensures that the work required grows logarithmically with the size of the list, leading to an overall time complexity of **O(n log n)**. This is much more efficient than processing the list as a whole, as it minimizes the number of comparisons and swaps needed.

The worst case being when the pivot consistently picks a value at one extreme (either the smallest or largest), leading to unbalanced partitions   
  
Bad because when the pivot consistently picks a value at one extreme, one of the partitions will contain almost all of the elements, while the other partition will be very small or empty. This results in the algorithm making very little progress at each recursion, requiring a linear number of comparisons for each level of recursion. As a result, the time complexity becomes **O(n²)**, which is much less efficient than the optimal **O(n log n)**, especially for large lists.

Now we have our pivot value chosen, the next step, is to split the list into three lists, what we will call, Left, Middle , Right

A for loop of length of number\_list [ an updating value within this function ] is used to populate the lists depending on < , > , = , pivot value

# empty lists for left, right, and middle / same as pivot

left = []

right = []

middle = []

# Loop through the number\_list and assign values to left, right, or middle

for x in number\_list:

if x < pivot:

left.append(x) # Add to left if x is less than pivot

elif x > pivot:

right.append(x) # Add to right if x is greater than pivot

else:

middle.append(x) # Add to middle if x is equal to pivot

Then a for loop is used to populate the lists depending in < , > , = , pivot value

Now we implement recursion on the left and right lists , merging with the middle starting pivot when done.

return quicksort(left) + middle + quicksort(right)

The left and right lists here are recursively called, until they return a num list length 1

if len(number\_list) <= 1:

At which point it has reached the smallest value for the left, and the smallest value for the right,

The call stack condition is then met, of len = 1 and it begins to return the value of a sorted list building from left to right, smallest to largest, one step at a time

until it reaches the top level of the stack, at which point the final left list is in order , and right

list is in order recombining them we get the fully sorted list which we then print to show the difference

print("After sorting:", sorted\_list)