Question 1
I)
Starting Array: [25, 17, 31, 13, 2, 40, 19]
i=-1, j=7, pivot=25

Decrement j (line 6-8) until j==6
Increment i (line 9-11) until i==0
swap [19, 17, 31, 13, 2, 40, 25]

Decrement j (line 6-8) until j==4
Increment i (line 9-11) until i==2
swap [19, 17, 2, 13, 31, 40, 25]

Decrement j (line 6-8) until j==3
Increment i (line 9-11) until i==4
swap [19, 17, 2, 13, 31, 40, 25]

return j = 4

ii)
assign pivot to A[low] – 1 operation
assign I – 1 operation
assign j – 1 operation
4 decrements + 4 comparisons = 8 operations
4 increments + 4 comparisons = 8 operations
two swaps * 3 assignments = 6 operations
comparison between I and j before each swap – 2 operations

27 operations total

iii) Worst case complexity here is O(n), where every element in the list would need to be compared and swapped as the loop would continue to run and trigger a swap operation for every element in the list.

iv) When a list is already sorted in quicksort, it results in the worst case complexity of O(n) as the algorithm will run through and check every element against the first element, but no swaps would occur.

Question 2
I)
Starting array: [25, 5, 10, 30, 15, 20]
The first non-leaf node is 10, which is smaller than it's child 20
The next non-leaf node is 5, which is smaller than it's children 30 and 15
The next node is 25, which needs to be heapified down. We swap it with it's smaller child 5
Array: [5, 25, 10, 30, 15, 20]
25 is still larger than one of it's children, so it swaps places with node 15
Array: [5, 15, 10, 30, 25, 20]

At each step, we ensure that any given node x is smaller than the values of x's children. This is the heap property. In a max heap, this would be reversed so that any node has children of smaller value than it.

ii)
Array: [5, 15, 10, 30, 25, 20]
8 is inserted as a new leaf node at the end of the array
Array: [5, 15, 10, 30, 25, 20, 8]
8's parent 10 is larger than it, so they swap
Array: [5, 15, 8 30, 25, 20, 10]
8's parent 5 is smaller than it, so we now have a correctly balanced min heap

iii)
Array: [5, 15, 8 30, 25, 20, 10]
Delete the root node by swapping it with the right-most leaf node.
Array: [10, 15, 8 30, 25, 20, 5]
5 now gets deleted
Array: [10, 15, 8 30, 25, 20]

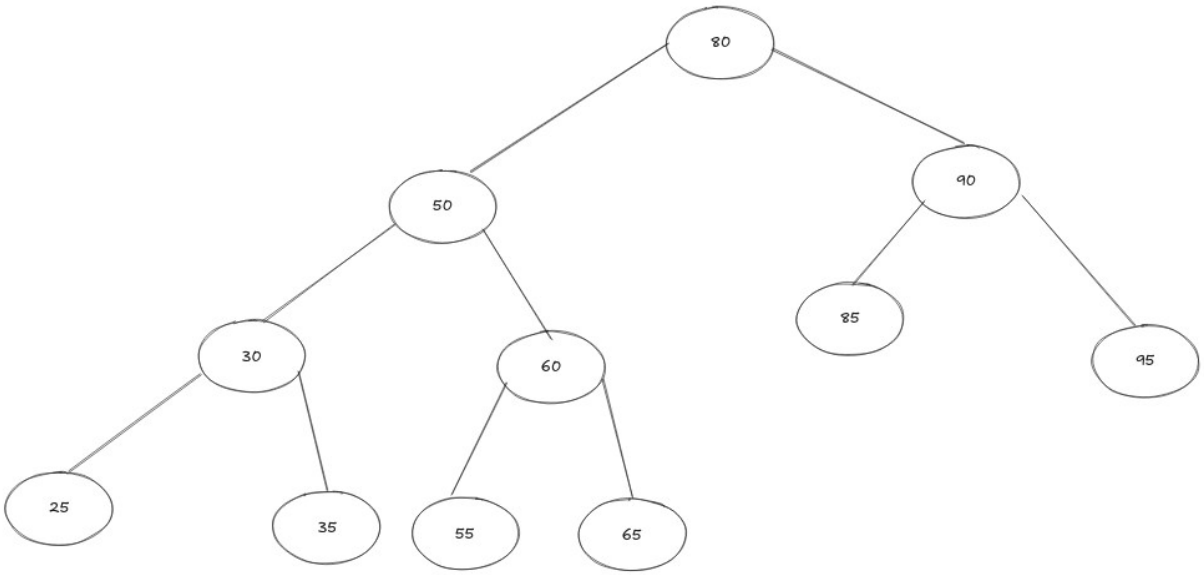
We now need to heapify down the new root node. 10 is larger than it's smaller child 8, so we swap those values
Array: [8, 15, 10 30, 25, 20]

10 is now larger than it's child node of 20, we have found it's correct place

iv)
Unsorted array – search O(n), insertion O(1), deletion O(n) as we need to move every element in memory
Sorted array – min element search O(1), insertion O(n) to find the correct location, deletion O(n) as we still need to move every element in memory
min heap – search O(log n), insertion and deletion are O(1), however they also trigger the heapify process which is O(log n). Overall a faster solution.

v) I would use a priority queue, which is often modelled on a min heap, as the heapify process should be able to handle all the required processes. The benefits of O(log n) reorganising (heapify) when new jobs get added or deadlines change make for faster operations than some other data structures mentioned – in this use case, a hash table makes little sense as you would basically need to already know what jobs you're looking for, whereas a binary search tree is inflexible for sorted data and would need resorting when priorities change.

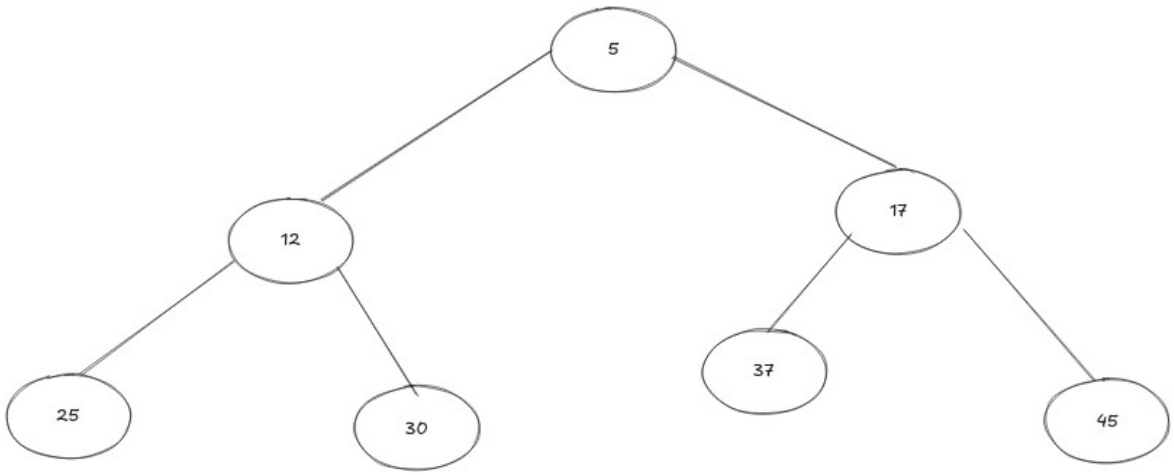
Question 3
I)



This tree has a height of 4 and 11 nodes

ii) Inorder traversal as an array:
[25, 30, 35, 50, 55, 60, 65, 80, 85, 90, 95]
One real-world use case of in order traversal is to sort the data in a tree.

Iii)

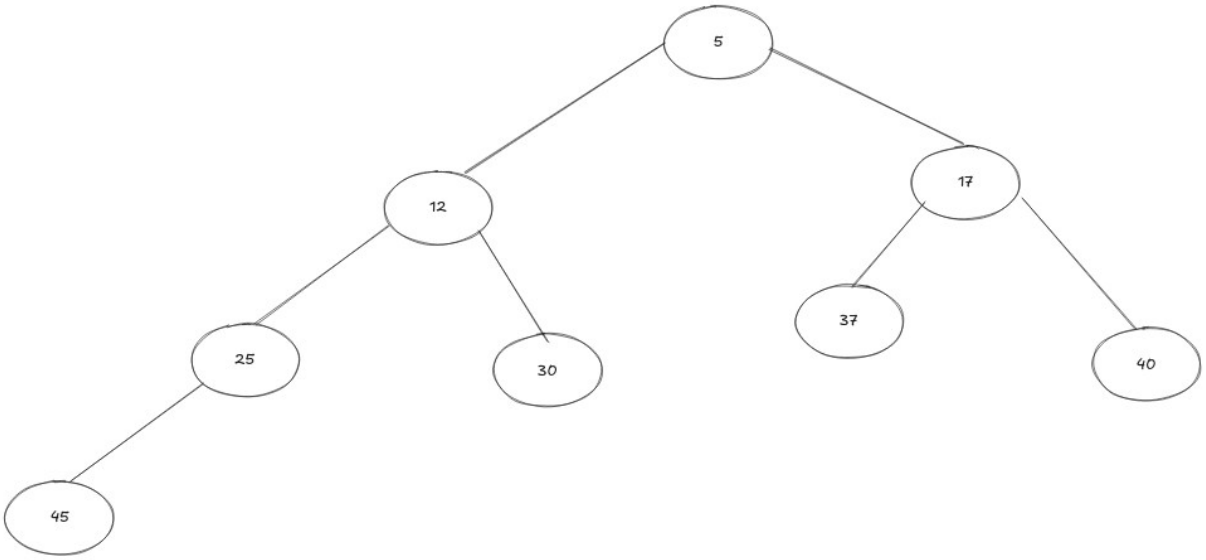


Searching for the value `17` can be done by starting at the root node and comparing every node using breath first search

start: 5
5 < 17
left node of 5: 12
12 < 17
right node of 5: 17
17 == 17
CONTINUED ON NEXT PAGE

iv)
Starting BST: [5, 12, 17, 25, 30, 37, 45]
Insertion of 40 starts at the end: [5, 12, 17, 25, 30, 37, 45, 40]
40 then rotates up the tree and down to it's correct location as a child of 17, pushing current node 45 up to the root and down to the other side of the tree to rest as the child of node 25.

The final tree should look like this:



Question 4

I) I'd suggest a Ring Buffer, which is often implemented as a form of linked list where the last element contains a pointer pointing to the first element. This steps from task to task in constant time, including from the last task back to the first, and will allow for insertion or deletion at any point by just updating the pointers that point to the next element to instead point to a new element. Assuming you want to insert in front of the current element, this also happens in constant time.

ii)

```
function NextTask(current)
      if current->next is not none:
            return current→next
      else:
            return current->prev
end function
```

Assuming the other functions on this ADT have been implemented correctly, this should never trigger the else block. However, to ensure that this doesn't just crash if the ring buffer is malformed, we instead end up infinitely rerunning the final two steps. This is where we could insert some error checking, such as assertions, which would result in a more manageable crash anyway.

Here we're assuming we have a doubly linked list, but most often a circular buffer is only a singly linked list

iii) The time complexity of stepping around the ring buffer is O(1) per step, or O(n) to iterate around the whole task management system. Taking into consideration the added next and prev pointers that are being stored, we end up taking a space complexity of O(3n), although as I mentioned previously, it's usually implemented as a singly linked list, so we only need to take into consideration a single pointer instead of two – so O(2n)

Question 5

I)

Adjacency matrix:

	D1	D2	D3	D4	D5
D1	0	5	2	9	-1
D2	5	0	3	-1	6
D3	2	3	0	4	2
D4	9	-1	4	0	2
D5	-1	6	2	2	0

Adjacency list:

{
D1: [(D2: 5), (D3, 2), (D4, 9)],
D2: [(D1, 5), (D3, 3), (D5, 6)],
D3: [(D1, 2), (D2, 3, (D4, 4), (D5,2)],
D4: [(D1, 9), (D3, 4), (D5, 2)],
D5: [(D2, 6), (D3, 2), (D4, 2)]
}

ii) A path can be constructed without ending up back at the starting point, with the shortest path taking you in the path of 1-3-4-5-2 at a distance of 14KM, although you could easily follow edge 2-1 back to the starting node of 1. Otherwise, this will construct a cycle. This satisfies Dirac's theorem by showing that every node is reachable by at least two other nodes – as 2 is the result of n / 2 in this graph.

Iii)

```
Algorithm FIND_ROUTE(G, source):
    beingVisited = stack()
    beingVisited.push(source)

    visited = [False for elem in G]
    visited[source] = True

    while beingVisited is not empty:
        current = beingVisited.pop()
        process(current)

        for each adjacent node [V] to current:
            if not visited[V]:
                beingVisited.push(V)
                visited[V] = True
            end if
        end for
    end while
end Algorithm
```

This function takes in the graph and a starting node, then uses depth first search and a stack to find the furthest path that can be taken in the graph.

iv)

v) The worst time complexity of my FIND_ROUTE algorithm is O(v+e), where v is the number of vertices and e is the number of edges to check on every recursive call. There's an additional spacial complexity of O(r) where r is the number of recursive calls you make, which is likely to be O(n^2).