DSA Coursework 2 23011124

data keys: 33, 30, 2, 52, 10, 14, 59, 80, 89, 96

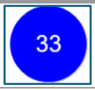
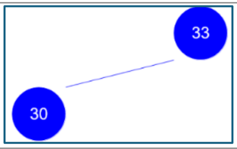
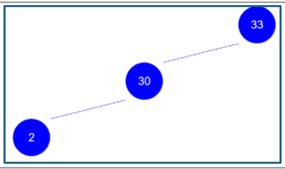
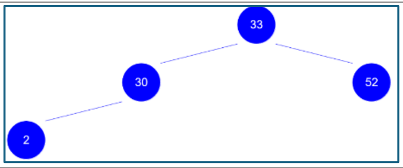
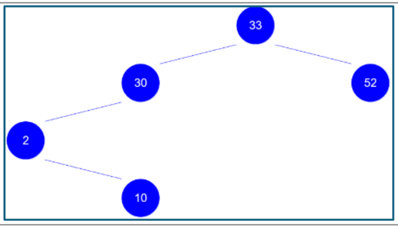
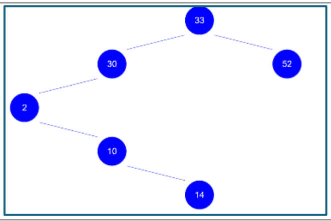
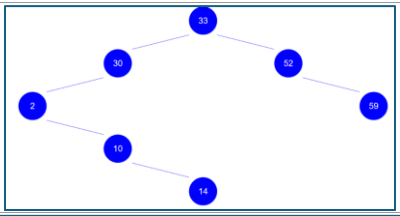
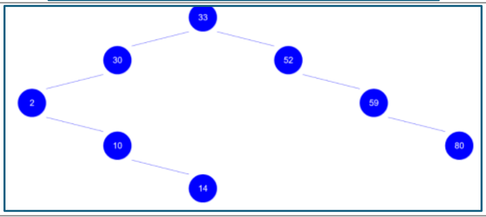
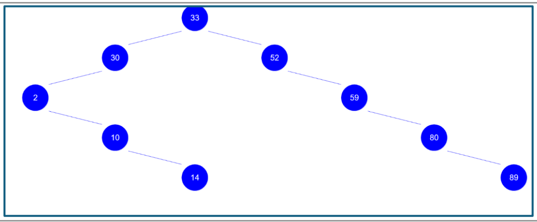
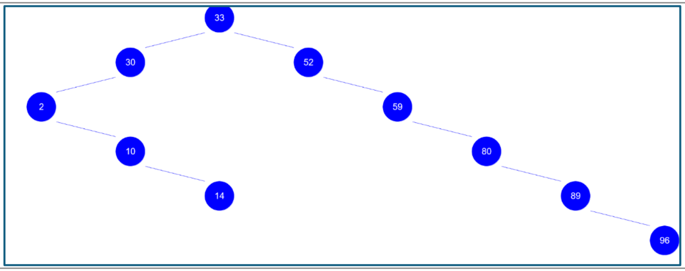
Q1-1. Stack Operations

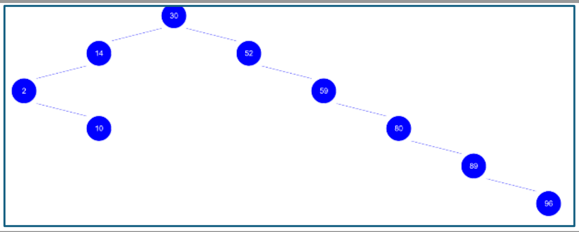
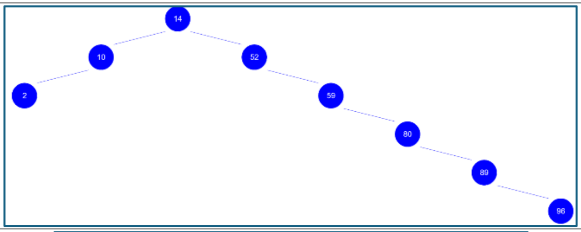
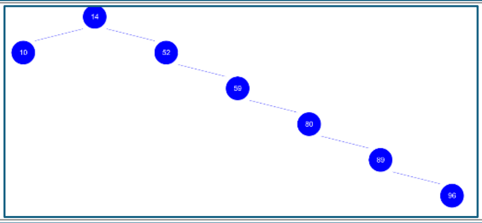
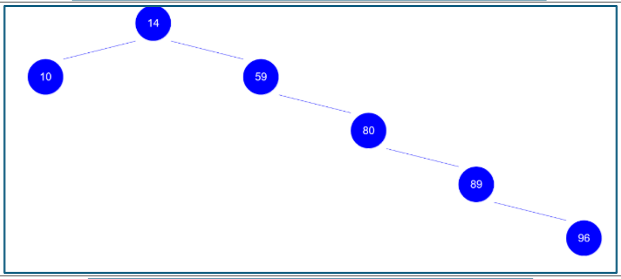
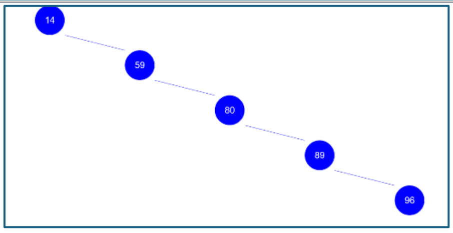
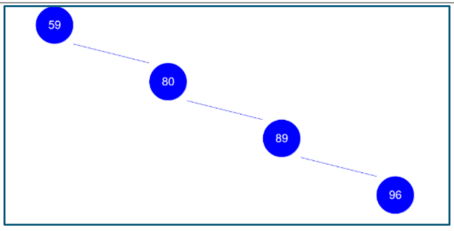
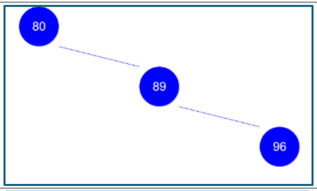
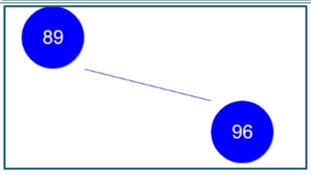
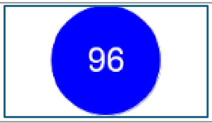
Operation	Input	Output	Stack Info	Explanation
push(x0) = push(33)	33		Top - 33, List - [33]	Push 33 on to the empty stack, 33 should now be at the top
push(x1) = push(30)	30		Top - 30, List - [33, 30]	Push 30 on to the stack, 30 should now be at the top
push(x2) = push(2)	2		Top - 2, List - [33, 30, 2]	Push 2 on to the stack, 2 should now be at the top
peek()		2	Top - 2, List - [33, 30, 2]	Peek at the top of the stack, 2 is on top so should be the element returned
pop()		2	Top - 30, List - [33, 30]	Pop from the top of the stack, 2 was on top so is removed and returned, stack is changed, top moves to previous element
pop()		30	Top - 33, List - [33]	Pop from the top of the stack, 30 was on top so is removed and returned, stack is changed, top moves to previous element
push(x3) = push(52)	52		Top - 52, List - [33, 52]	Push 52 on to the stack, 52 should now be at the top
push(x4) = push(10)	10		Top - 10, List - [33, 52, 10]	Push 10 on to the stack, 10 should now be at the top
size()		3	Top - 10, List - [33, 52, 10]	Count the amount of elements in the stack, there are 3 so 3 should be returned
pop()		10	Top - 52, List - [33, 52]	Pop from the top of the stack, 10 was on top so is be removed and returned, stack is changed, top moves to previous element
isEmpty()		False	Top - 52, List - [33, 52]	Check if the stack is empty or not, there are 2 elements so it shouldn't flag as empty

Q1-2. Queue Operations

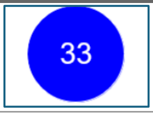
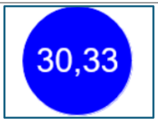
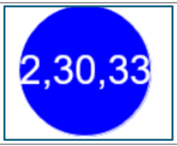
Operation	Input	Output	Queue Info	Explanation
enqueue(x5) = enqueue(14)	14		Front - 14, List - [14]	Enqueue 14 into the empty queue, 14 should be at the front
enqueue(x6) = enqueue(59)	59		Front - 14, List - [14, 59]	Enqueue 59 at the end of the queue, 14 should still be at the front
enqueue (x7) = enqueue(80)	80		Front - 14, List - [14, 59, 80]	Enqueue 80 at the end of the queue, 14 should still be at the front
isEmpty()		False	Front - 14, List - [14, 59, 80]	Check if the queue is empty or not, there are 3 elements so it shouldn't flag as empty
front()		14	Front - 14, List - [14, 59, 80]	Look at the front of the queue, 14 is at the front so should be the element returned
dequeue()		14	Front - 59, List - [59, 80]	Dequeue from the front of the queue, 14 was at the front so is removed and returned, queue is changed, front moves to next element (59)
dequeue()		59	Front - 80, List - [80]	Dequeue from the front of the queue, 59 was at the front so is removed and returned, queue is changed, front moves to next element (80)
enqueue (x8) = enqueue(89)	89		Front - 80, List - [80, 89]	Enqueue 89 at the end of the queue, 80 should still be at the front
enqueue (x9) = enqueue(96)	96		Front - 80, List - [80, 89, 96]	Enqueue 96 at the end of the queue, 80 should still be at the front
size()		3	Front - 80, List - [80, 89, 96]	Count the amount of elements in the queue, there are 3 so 3 should be returned
dequeue()		80	Front - 89, List - [89, 96]	Dequeue from the front of the queue, 80 was at the front so is removed and returned, queue is changed, front moves to next element (89)
isEmpty()		False	Top - 52, List - [33, 52]	Check if the queue is empty or not, there are 2 elements so it shouldn't flag as empty

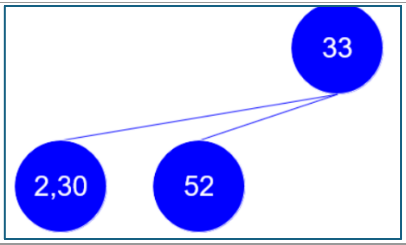
Q2-1. Binary Search Tree

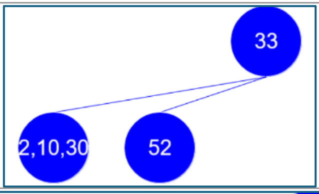
Operation	Tree	Explanation
insert(x0) = insert(33)		Insert 33 at the root of the tree
insert(x1lx0) = insert(30l33)		Insert 30 as left child of 33 as 30 < 33
insert(x2lx1) = insert(2l30)		Insert 2 as left child of 30 as 2 < 30
insert(x3rx2) = insert(52r33)		Insert 52 as right child of 33 as 52 > 33
insert(x4rx2) = insert(10r2)		Insert 10 as right child of 2 as 10 > 2
insert(x5rx6) = insert(14r10)		Insert 14 as right child of 10 as 14 > 10
insert(x6rx3) = insert(59r52)		Insert 59 as right child of 52 as 59 > 52
insert(x7rx6) = insert(80r59)		Insert 80 as right child of 59 as 80 > 59
insert(x8rx7) = insert(89r80)		Insert 89 as right child of 80 as 89 > 80
insert(x9rx8) = insert(96r89)		Insert 96 as right child of 89 as 96 > 89

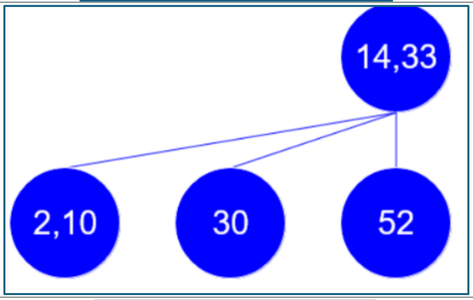
Operation	Tree	Explanation
delete(x0) = delete(33)		Delete 33 from the root of the tree, moving 30 into the root space as it's the direct predecessor of 33 and moving 14 into the hole left by 30 as it's the next direct predecessor
delete(x1) = delete(30)		Delete 30 from the root of the tree, moving 14 into the root space as it's the direct predecessor of 30 and moving 10 into the hole left by 14 as it's the next direct predecessor
delete(x2) = delete(2)		Delete 2 from the bottom of the tree, since it's a leaf, nothing else needs to be moved
delete(x3) = delete(52)		Delete 52 from the right child of the root of the tree, moving 59 then 80 then 89 then 96 all up into the space left by the previous number as they are the successors of 52
delete(x4) = delete(10)		Delete 10 from the bottom of the tree (left child of root), since it's a leaf, nothing else needs to be moved
delete(x5) = delete(14)		Delete 14 from the root of the tree, moving 59 then 80 then 89 then 96 all up into the space left by the previous number as they are the successors of 14
delete(x6) = delete(59)		Delete 59 from the root of the tree, moving 80 then 89 then 96 all up into the space left by the previous number as they are the successors of 59
delete(x7) = delete(80)		Delete 80 from the root of the tree, moving then 89 then 96 all up into the space left by the previous number as they are the successors of 14
delete(x8) = delete(89)		Delete 89 from the root of the tree, moving 96 up into the space left by 89 as it is the direct successor
delete(x9) = delete(96)		Delete 96 from the root of the tree

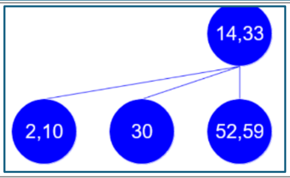
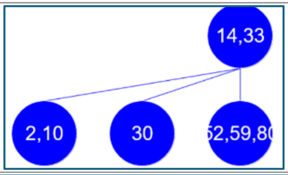
Q2-2. 2-4 Tree

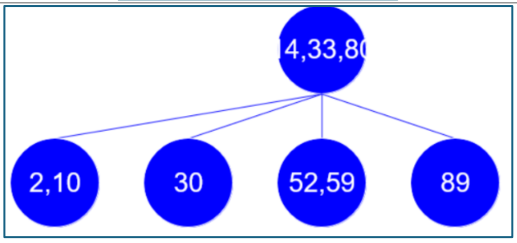
Operation	Tree	Explanation
insert(x0) = insert(33)		Insert 33 at the root of the tree
insert(x1) = insert(30)		Insert 30 at the root of the tree before (to the left of) 33
insert(x2) = insert(2)		Insert 2 at the root of the tree before (to the left of) 30

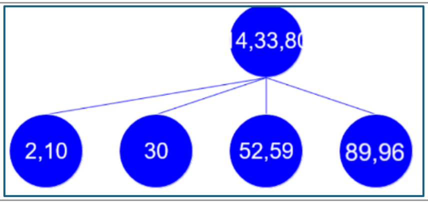
insert(x3) = insert(52)
split(x0) = split(33)
		Insert 52 at the root of the tree after (to the right of) 33, however root was full;
		Split tree via 33, bringing 33 back to root and [2,30] as left child, 52 as right child of root

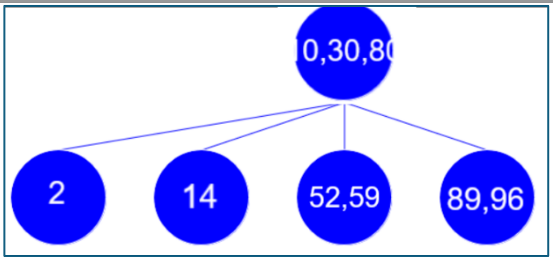
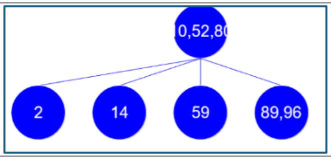
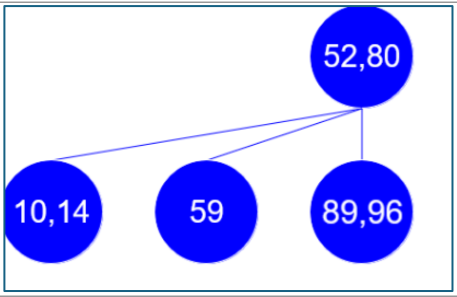
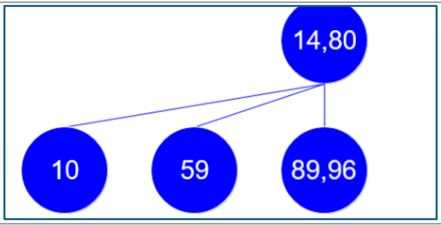
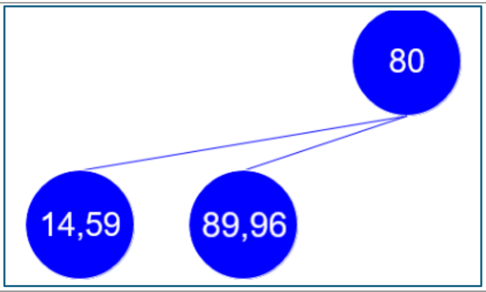
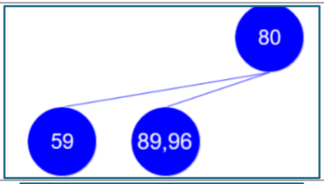
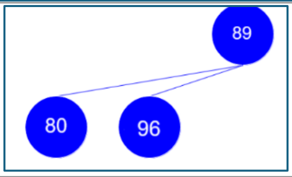
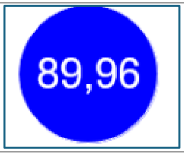
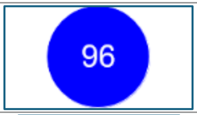
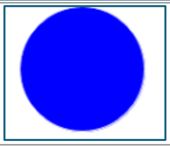
insert(x4) = insert(10)		Insert 10 at the left child of the tree before (to the left of) 30 and after (to the right of) 2

insert(x5) = insert(14)
split(x5) = split(14)
		Insert 14 at the left child of the tree before (to the left of) 30 and after (to the right of) 10, however left child was full;
		Split tree via 14, bringing 14 to the root and [2,10] as left child, 30 as middle child and 52 as right child of root

insert(x6) = insert(59)		Insert 59 at the right child of the tree after (to the right of) 52
insert(x7) = insert(80)		Insert 80 at the right child of the tree after (to the right of) 59

insert(x8) = insert(89)
split(x7) = split(80)
		Insert 89 at the right child of the tree after (to the right of) 80, however right child was full;
		Split tree via 80, bringing 80 to the root and [2,10] still as left child, 30 as left-middle child, [52,59] as right-middle child, and 89 as right child of root

insert(x9) = insert(96)		Insert 96 at the right child of the tree after (to the right of) 89

Operation	Tree	Explanation
delete(x0) = delete(33)		Delete 33 from the root of the tree, moving 30 into the root space as it's the direct predecessor of 33 and moving 14 into the hole left by 30 as it's the next direct predecessor and moving 10 into the hole left by 14 in the root space as it's the next direct predecessor
delete(x1) = delete(30)		Delete 30 from the root of the tree, moving 52 into the root space as it's the direct successor of 30
delete(x2) = delete(2)		Delete 2 from the left child of the tree, it's in a leaf node but it's the only key in the node and the left-middle node (right sibling) also only has one key, so node deleted and 10 from root node will merge into new left child with 14, everything else stays the same then
delete(x3) = delete(52)		Delete 52 from the root of the tree, moving 14 into the root space left by 52 as it's the direct predecessor
delete(x4) = delete(10)		Delete 10 from the left child of the root of the tree, it's in a leaf node but it's the only key in the node and the middle node (right sibling) also only has one key, so node deleted and 14 from root node will merge into new left child with 59, everything else stays the same then
delete(x5) = delete(14)		Delete 14 from the left child of the root of the tree, nothing else needs to change as it was in a leaf with 2 keys previously
delete(x6) = delete(59)		Delete 59 from the left child of the root of the tree, since sibling has 2 keys, can rotate tree so 80 is now left child, 89 is now root and 96 is now right child
delete(x7) = delete(80)		Delete 80 from the left child of the root of the tree, moving 96 into the root that previously only contained 89
delete(x8) = delete(89)		Delete 89 from the root of the tree
delete(x9) = delete(96)		Delete 96 from the root of the tree

Q3.1 Dijkstra's algorithm application

SHORTEST DISTANCE

Step	Node	Distance	Path	Step	Node	Distance	Path
1	A	0	-	4	D	26	A -> F -> E -> D
2	F	2	A -> F	5	C	30	A -> C
3	E	12	A -> F -> E	6	B	33	A -> B

Step 1:

Shortest Distance -> {A: 0}, Unvisited -> {B: ∞, C: ∞, D: ∞, E: ∞, F: ∞}

Step 2:

Shortest Distance -> {A: 0, F: 2}, Unvisited -> {B: 33, C: 30, D: ∞, E: ∞}, Calculations -> d(A,B) = 33; d(A,C) = 30; d(A,F) = 2

Step 3:

Shortest Distance -> {A: 0, F: 2, E: 12}, Unvisited -> {B: 33, C: 30, D: 98} Calculations -> d(A,B) = d(A,F) + d(F,B) = 2 + 80 = 82 > 33; d(A,D) = d(A,F) + d(F,D) = 2 + 96 = 98 < ∞; d(A,E) = d(A,F) + d(F,E) = 2 + 10 = 12 < ∞

Step 4:

Shortest Distance -> {A: 0, F: 2, E: 12, D: 26}, Unvisited -> {B: 33, C: 30} Calculations -> d(A,C) = d(A,F) + d(F,E) + d(E,C) = 2 + 10 + 52 = 64 > 30; d(A,D) = d(A,F) + d(F,E) + d(E,D) = 2 + 10 + 14 = 26 < 98

Step 5:

Shortest Distance -> {A: 0, F: 2, E: 12, D: 26, C: 30}, Unvisited -> {B: 33} Calculations -> d(A,C) = d(A,F) + d(F,E) + d(E,D) + d(D,C) = 2 + 10 + 14 + 89 = 117 > 30

Step 6:

Shortest Distance -> {A: 0, F: 2, E: 12, D: 26, C: 30, B: 33}, Unvisited -> {} Calculations -> d(A,B) = d(A,C) + d(C,B) = 30 + 59 = 89 > 33

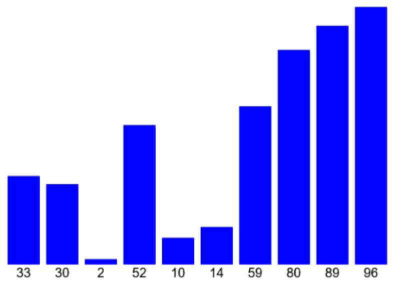
KNOWN DISTANCE AT STEP

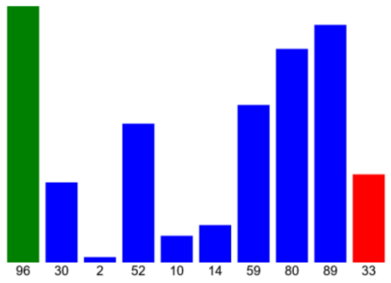
	1	2	3	4	5	6
A	0					
B	∞	33	33	33	33	33
C	∞	30	30	30	30	
D	∞	∞	98	26		
E	∞	∞	12			
F	∞	2				

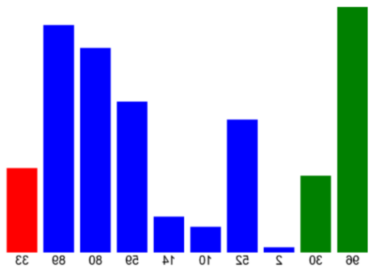
Q3.2 Complexity explanation

Dijkstra's algorithm exhibits a time complexity of $O(V^2)$ when implemented with a simple adjacency matrix, while the use of a priority queue reduces it to $O((V + E) \log V)$ (Cormen et al., 2009; Mehlhorn and Sanders, 2008). The space complexity is typically $O(V + E)$ when using adjacency lists for graph representation. Advanced data structures like Fibonacci heaps can further improve the time complexity to $O(V \log V + E)$, particularly beneficial for dense graphs (Barbehenn, 1998).

Q4.1 Quicksort



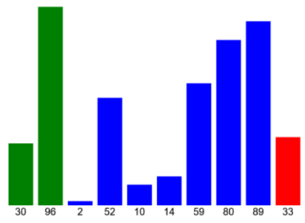




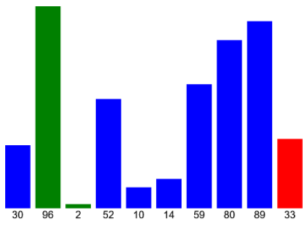
Step 0: Initial array

Step 1a: Comparing 96 with pivot 33

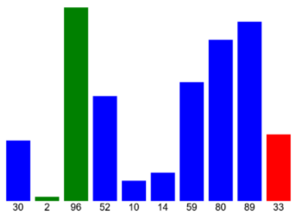
Step 1b: Comparing 30 with pivot 33



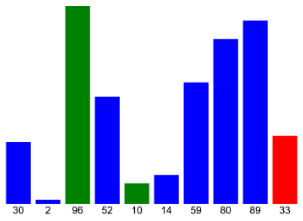
1c: Swapped 30 and 96



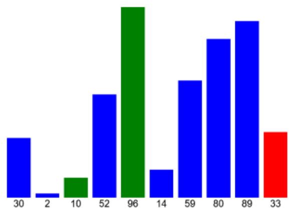
1d: Comparing 2 with pivot 33



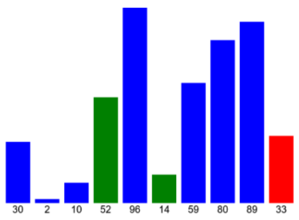
1e: Swapped 2 and 96



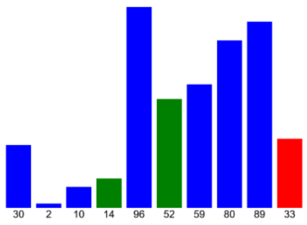
1f: Comparing 10 with pivot 33



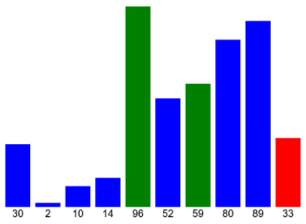
1g: Swapped 10 and 96



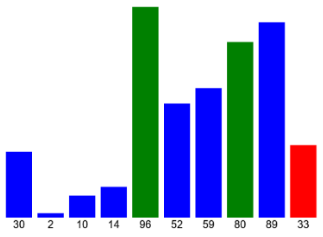
1h: Comparing 14 with pivot 33



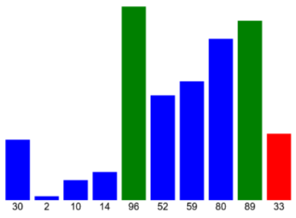
1i: Swapped 14 and 52



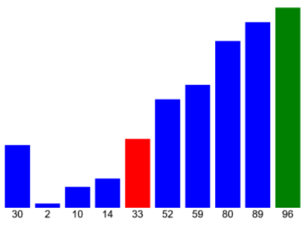
1j: Comparing 59 with pivot 33



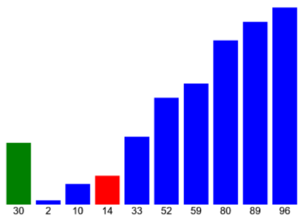
1k: Comparing 80 with pivot 33



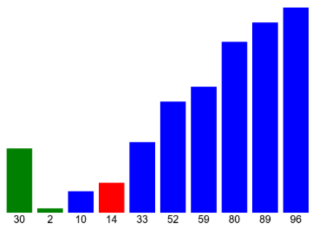
1l: Comparing 89 with pivot 33



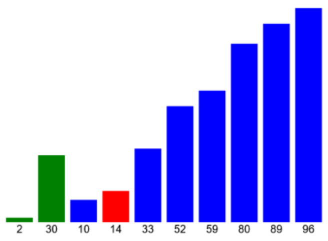
1m: Swapped 96 and pivot 33



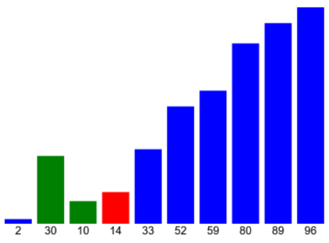
2a: Comparing 30 with pivot 14



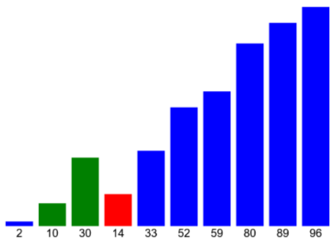
2b: Comparing 2 with pivot 14



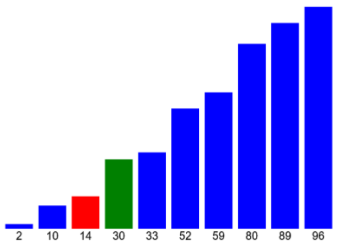
2c: Swapped 2 and 30

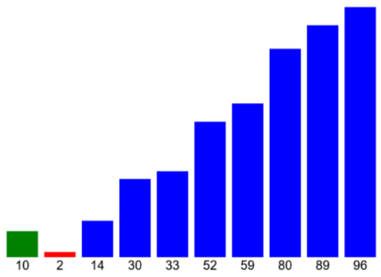


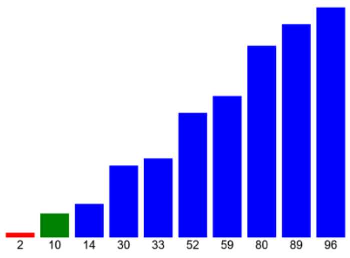
2d: Comparing 10 with pivot 14



2e: Swapped 10 and 30







2f: Swapped 30 and pivot 14

Step 3a: Comparing 10 with pivot 2

3b: Swapped 10 and pivot 2

Q4.2 Complexity explanation

Quicksort has an average-case time complexity of O(n log n) due to its divide-and-conquer strategy, but in the worst case, where the pivot selection is poor, it degrades to $O(n^2)$ (Cormen et al., 2009; Hoare, 1962). Its space complexity is O(log n) on average due to recursive stack usage, though this can increase to O(n) in the worst case if recursion is not optimized (Mehlhorn and Sanders, 2008).

References:

- Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., 2009. Introduction to Algorithms. 3rd ed. MIT Press.

- Hoare, C.A.R., 1962. Quicksort. The Computer Journal, 5(1), pp.10–15.

- Mehlhorn, K. and Sanders, P., 2008. Algorithms and Data Structures: The Basic Toolbox. Springer.

- Barbehenn, M., 1998. A Note on the Complexity of Dijkstra's Algorithm. Available at: [ResearchGate](https://www.researchgate.net/publication/3043930).