1. Consider the function `my_function(L, tol)` where `L` is assumed to be a Python list containing floating point numbers, and `tol` is assumed to be a single floating point number.

```
1 def my_function(L, tol):
2     for j in range(len(L)):
3         for k in range(j+1, len(L)):
4             if abs(L[j] - L[k]) <= tol:
5                 return True
6     return False
```

   (a) What is this function doing?

   (b) What is its worst case asymptotic running time as a function of `n = len(L)`?

   (c) Give an asymptotically faster equivalent implementation of `my_function`.

(5+5+5)

(a) The function checks if any two entries of L are identical up to a difference up to tol (as absolute value in floating point). In that case, it returns true, otherwise false.

(b) If no matching pair is found, it will need to check

$$(n-1) + (n-2) + \dots + 1 = O(n^2)$$

pairs. So the worst-case asymptotic running time is $O(n^2)$

(c) 1. Sort the list at a cost of $O(n \log n)$

2. Check only nearest neighbor pairs at a cost of $O(n)$

$\Rightarrow$ Total cost is $O(n \log n)$.

2. You are given that the *pre-order* traversal of a binary tree yields the sequence A–B–D–E–C–F, while the *in-order* traversal of the same tree yields the sequence D–B–E–A–F–C.

   (a) Reconstruct the binary tree given this information.

   (b) State the *post-order* traversal sequence of the same tree.

   (c) Can you always reconstruct a binary tree if you know the pre-order and the in-order traversal sequence? Explain why or give a counter example.

(5+5+5)

(a) A must be root (first in pre-order traversal)

⇒ left subtree of A is D-B-E (before A in in-order trav.)

⇒ B is root of that subtree (first in pre-order trav.)

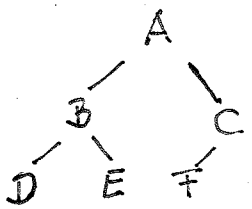
⇒ left subtree of A is B via in-order trav.

⇒ right subtree of A is F-C (after A in in-order trav.)

⇒ this subtree is rooted at C (first in pre-order)

⇒ right subtree of A is C (via in-order)

Full tree:



(b) Post-order traversal:

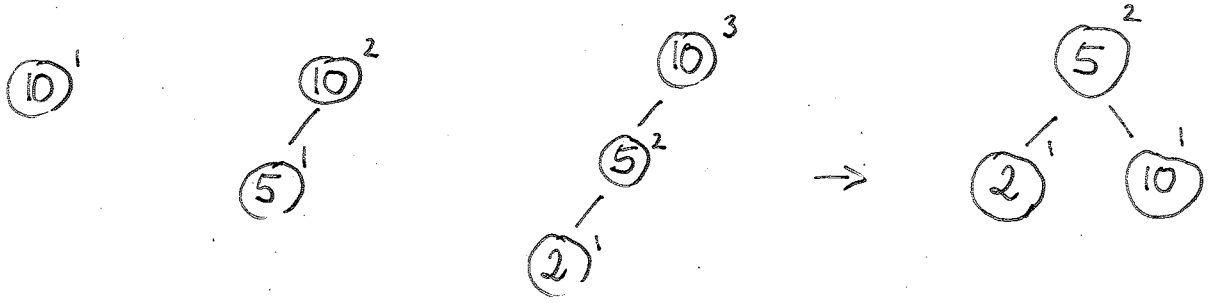
D-E-B-F-C-A

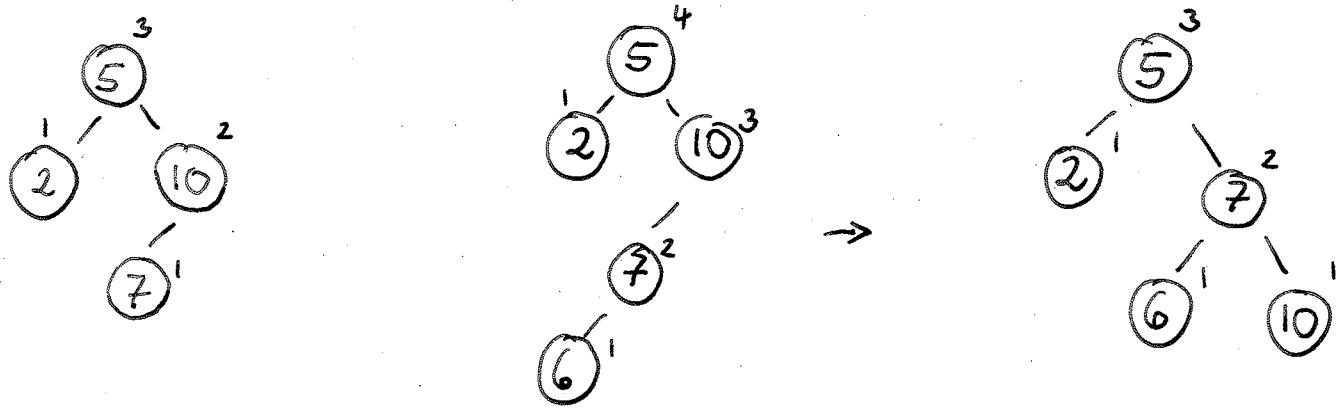
(c) Yes:
- First element in pre-order is root
- The root in in-order separates left and right subtrees
- Restrict the traversals to each subtree and proceed recursively. This will deterministically determine the entire tree.

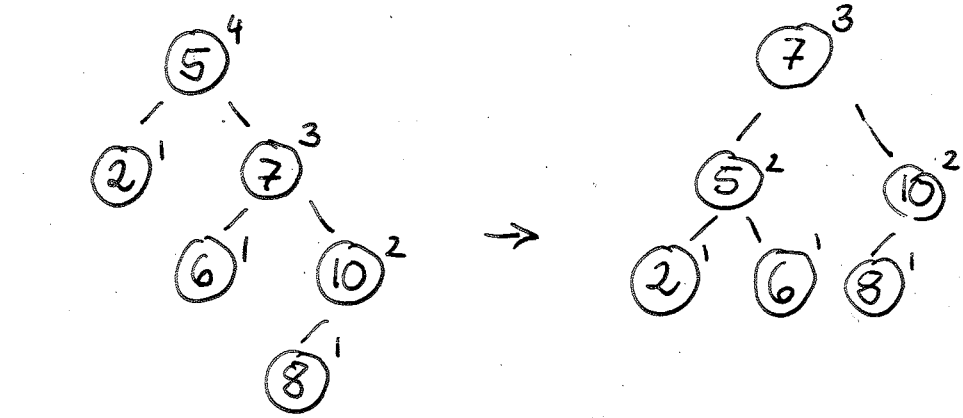
3. Insert the following sequence of elements into an initially empty AVL tree. Indicate all subtree heights and show the changes introduced by the insertions and the subsequent rebalancing step-by-step. (You may consult the summary of tree rotations on the front page.)

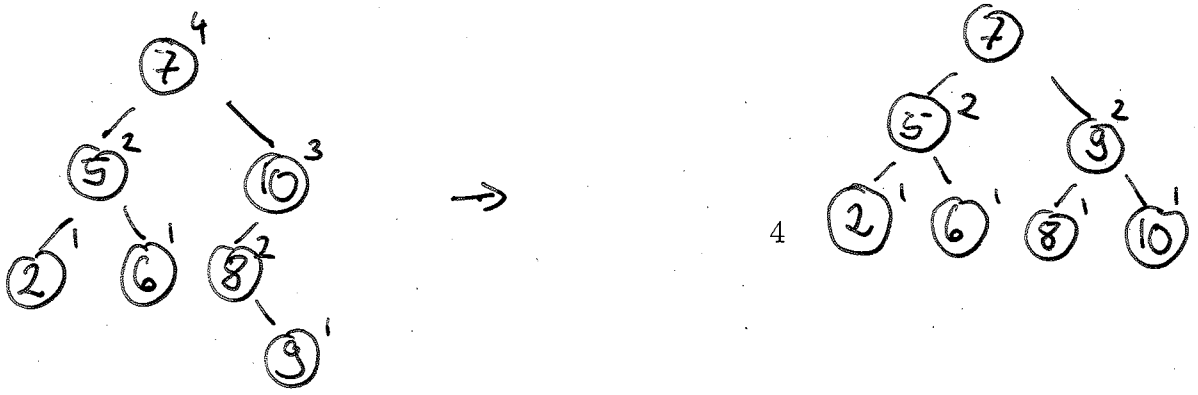
$$10, 5, 2, 7, 6, 8, 9$$

(10)









4. State one distinct advantage that would favor each of the following sorting algorithms for a specific use case.

   (a) Insertion sort
   (b) Quick sort
   (c) Merge sort
   (d) Heap sort
   (e) Splay sort

(5)

(a) Fast on short sequences due to small overhead, fast on almost sorted sequences. (In general, however, $O(n^2)$, thus not competitive.)

(b) Simple algorithm that is $O(n \log n)$ on average which is very fast in practice. (However, worst case is $O(n^2)$!)

Good general-purpose, if performance guarantees are not required.

(c) $O(n \log n)$ guaranteed, good cache locality, stable sort.

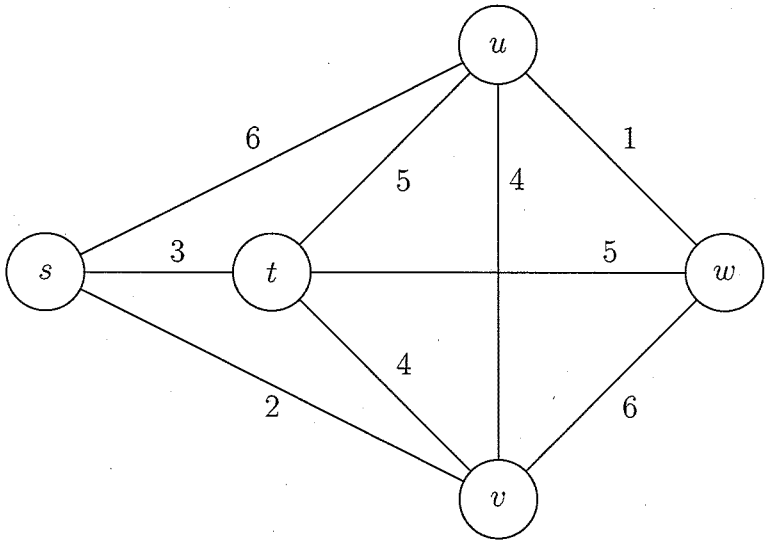
(But hard to implement in-place)

(d) $O(n \log n)$ guaranteed, easy in-place sorting

(But poor cache locality)

(e) $O(n \log n)$ guaranteed, in general more difficult to implement and not as fast as heap or merge sort, but performs $O(n)$ on almost sorted sequences and adapts quasi-optimally to partially sorted sequences.

5. Consider the following weighted graph.

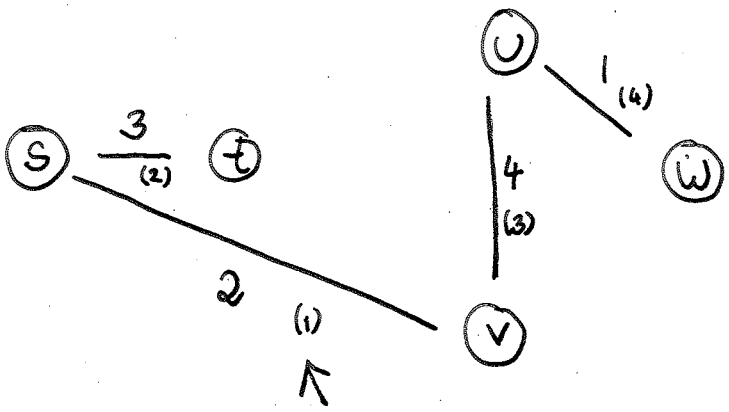


(a) Draw the minimum spanning tree that results from running the Prim–Jarník algorithm on this graph, starting at vertex s. List the order in which edges are added to the tree.

(b) Draw the minimum spanning tree that results from running Kruskal's algorithm on this graph. List the order in which edges are added to the tree.

(5+5)

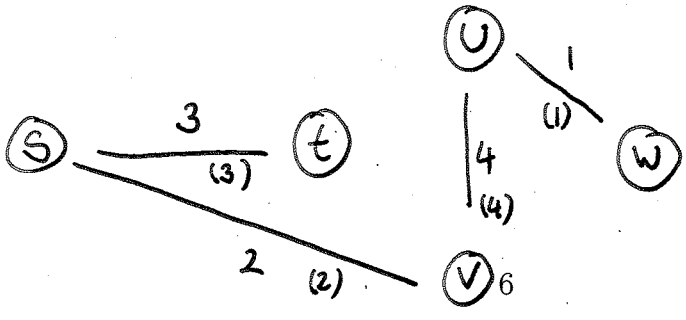
(a)



↖ step of Prim-Jarník

(So the trees co-incide, but edges are added in different order.)

(b)



↖ step of Kruskal

6. (a) State Dijkstra's algorithm for computing the shortest path in a weighted graph G with vertex set V, edge set E, and non-negative weights w from a starting vertex $s \in V$ to each vertex $v \in V$ using Python or pseudo-code. You may assume that you have access to an implementation of the Adaptable Priority Queue ADT.

   (b) Adaptable priority queues are often implemented using the heap data structure. State the asymptotic running times for each of the core methods of the Adaptable Priority Queue ADT. Give a brief justification in each case.

   (c) For Dijkstra's algorithm, there are situations where a simpler implementation of the Adaptable Priority Queue ADT is actually asymptotically faster. Explain!

(5+5+5)

(a)

```
Initialize: D[s] = 0
            D[v] = inf   for each v ∈ V, v ≠ s

Q = PriorityQueue();
for v ∈ V:
    Q.insert(v, D[v])        # D[v] is the key!

while Q.not_empty():
    u = Q.remove_min()
    for v in neighbors(u) ∩ Q:
        if D[u] + w(u,v) < D[v]:
            D[v] = D[u] + w(u,v)
            Q.update(v, D[v])
```


Now D[v] contains the shortest-path-length from s to v.

(Scratch paper)

(b) • remove_min : $O(\log n)$ as last element is promoted to root and might need to "bubble down" $\log n$ levels to restore heap order.

• insert : $O(\log n)$ as new element might need to "bubble up" $\log n$ levels to restore heap order.

• update : Assuming that the entry that needs updating can be located via direct handle or hash in $O(1)$ time, the update can cause "bubbling up" (or down) at again $O(\log n)$ cost (worst case).

(c) If the graph is (nearly) complete, the inner loop goes over $(n-1), (n-2), \ldots, 1$ elements, so the Q.update may be performed $O(n^2)$ times.

• Heap update: costs $\log n$, so total cost is $O(n^2 \log n)$

• Unordered list update: $O(1)$, so total cost of all updates is $O(n^2)$. In this case, Q.remove_min costs $O(n)$, so all remove-min operations also cost $O(n^2)$ for total cost $O(n^2)$

⇒ In this case, an unordered list implementation is preferred.