I) Fractional Knapsack Problem

Algorithm Fractional Knapsack
// Input: Items array with weight, value, and shelf life; maximum weight to take
// Output: Maximum value of items
Sort item, based on the parameter $\frac{\text{value}}{\text{weight} * \text{shelf life}}$ (in reverse order)

```
total_value = 0
for each item in items:
    if weight of item <= W:
        W = W - weight of item
        total_value = total_value + value of item
    else:
        fraction = W / weight of item
        total_value = total_value + (value of item) * fraction
        exit loop
return total_value
```

Time complexity:
1) Sorting items array takes O(n log n) time.
2) In the worst case, every item is processed in the for loop. This leads to a complexity of O(n) for n items.
3) Hence, total time complexity is O(n log n) + O(n). For large value of n it becomes O(n log n).

Knapsack Problem - Brute Force.

// Input: Items array with weight & value, maximum weight W

// Output: Maximum value.

1. Generate all subsets of items array
2. Set max-val = 0, max-subset = null
3. For each subset in subsets:

~~if $\sum_{i \in subset}$ weight(i) <= W and~~

~~max-val = max~~

if $\sum_{i \in subset}$ weight(i) <= W and $\sum_{i \in subset}$ value(i) > max-val:

max-val = $\sum_{i \in subset}$ value(i)

max-subset = subset

4. Return max-val, max-subset

Time complexity:

i) An array/set of length n has 2^n subset

ii) The for loop iterates over each 2^n subsets.

iii) In one iteration of for loop, it calculates weight & value of the subset which takes O(n) time.

iv) Hence time complexity is $O(n2^n)$

II) Huffman Encoding.

i) Getting character - frequency mapping
// Input - String of length L (Text)
// Output - Hashmap where key is character & value is frequency.
Declare hashmap of character, frequency
i.e. (char, int) type
for each character in Text:
    if character not in hashmap:
        hashmap.insert((character, 1))
    else:
        hashmap[character] += 1
return hashmap.

Time complexity : O(L) as we iterate over every character of text of length L

ii) Build Tree.
// Input : Hashmap of character frequencies
// Output : Huffman tree.
~~for character, frequency~~
Declare empty nodes array.
for character, frequency in hashmap
    nodes.add(Node(frequency, character))
heapify(nodes)
while nodes.length > 1:
    left = heappop(nodes)
    right = heappop(nodes)
    Declare new node z

```
z.left = left
z.right = right
z.freq = left.freq + right.freq
heappush(nodes, z)
return heappop(nodes)
```

Time Complexity

i) The min-heap is built in O(n) time using the heapify procedure

ii) ~~The while loop~~ Inside the while loop, two elements are removed & one element is inserted into the heap, until only 1 element is left in the heap.

iii) ~~The~~ Hence the inner operation executes ~~for~~ in O(~~n~~ log n) time & the loop executes for n-1 times. ~~Outside the loop, the heappop operation occu~~ This leads to an overall complexity of O(n log n) time.

iii) Get Codes (node, ~~current~~ val = "")
// Input : Root of huffman tree, character
'val' → default value empty
// Output : ~~D~~ Hashmap containing huffman
codes of each & character
if node is ~~Node None~~ null : node = root
if node → left is not null
Get Codes (node → left, val + node → huff)
if node → right is not null
Get Codes (node → right, val + node → huff)
if node → left & node → right are null
codes [node → char] = val + node → huff.

Time complexity : O(n), where n is the number of nodes in the tree. This is because the algorithm visits every node in the tree.

iv) Compress
// Input : String to be compressed
// Output : Compressed string via huffman
encoding
result = ""
for char in string:
result += codes [char]
return result

Time Complexity : O(n) where n is length of string. Loop iterates over each character & lookup in codes table is O(1).

v) Decompress.

```
// Input : Compressed string , Root of huffman tree
// Output : Decompressed string via
            huffman decoding.
result = ''
node = root of huffman tree
for bit in string:
        if bit is 0:
            node = node -> left
        else:
            node = node -> right
        if node -> left & node -> right are null:
            res += node -> char
            return
            node = root
return res
```

Time complexity

i) In the for loop, one or two operations occur depending on whether the second if statement is satisfied or not.

ii) Total time complexity is O(n) where n is length of compressed string.