1. In Bubble sort, we "bubble" large items to the back of the array by repeatedly comparing adjacent items. Following are two versions. The first argument, A, is an array of distinct positive integers and the second, k, is the size of the array. We assume that the array is indexed $A[0, \ldots\ldots k-1]$.

```
def BUBBLESORT_1(A):
    for i in range(1, len(A)):
        for j in range(len(A), i, −1):
            if A[j] < A[j+1]:
                k = A[j]
                A[j] = A[j+1]
                A[j+1] = k
    return A
```

```
def BUBBLESORT_2(L):
    swapped = True
    while swapped:
        swapped = False
        for i in range(len(L) − 1):
            if L[i] > L[i+1]:
                k = L[i]
                A[i] = A[i+1]
                A[i+1] = k
                swapped = True
```

```
def BUBBLESORT_3(A):
    if len(A) == 1:
        return A
    for i in range(1, len(A)−1):
        if A[i] > A[i+1]:
            swap(A[i], A[i+1])
    return BUBBLESORT_3(A[:len(A)−1]) + A[len(A)]
```

Assuming that it takes k bits to encode $min, \ldots, max$

(a) In $\Theta(\cdot)$ compare the run-time of the two algorithms
(b) What is the worst case space efficiency for the two algorithms

2. Recall our array encoding of a complete binary tree: the parent of a node at index i is at index $\lfloor i/2 \rfloor$, its left child is at index $2i$, and its right child is at index $2i+1$. We can use such an array encoding for any kind of binary tree, e.g., a binary search tree, or a binary heap. A binary search tree has the property that the key of a node is larger than every key in its left sub-tree, and smaller than every key in its right sub-tree. In a heap, the key of a node is smaller than the keys of all its descendants.

(a) Given as input an array $A[1, \ldots, n]$ that encodes a complete binary BST of distinct keys, write down pseudo-code for an efficient algorithm to convert $A[0, \ldots\ldots n-1]$ into a complete heap.
(b) Given as input an array $A[1, \ldots, n]$ that encodes a complete binary BST of distinct keys, write down pseudo-code for an efficient algorithm to convert $A[0, \ldots\ldots n-1]$ into a complete heap **BUT** this time in-place *i.e.* without creating any intermediate data structures to store the entirety of the tree

3. Recall the notion of a Minimum spanning tree for a weighted connected undirected graph $G = \langle V, E, w \rangle$ with all positive edge weights: it is a spanning tree that minimizes the sum of weights of edges across all spanning trees of G.

KRUSKAL($G = \langle V, E, w \rangle$)

$A \leftarrow \phi$
foreach $u \epsilon V$ do MAKE-SET(u)
foreach $\langle u, v \rangle \epsilon E$ in non decreasing order of weight
    if FIND-SET(u) $\neq$ FIND-SET(v)
        $A \leftarrow A \cup \langle u, v \rangle$
        UNION(U,V)
return A

(a) Prove correctness of Kruskal's Algorithm

(b) Show that the smallest non-cyclic edge is a valid greedy choice

(c) Let e be a maximum-weight edge on some cycle of connected graph $G = \langle V, E \rangle$. Prove that there is a minimum spanning tree of $G' = \langle V, E - e \rangle$ that is also a minimum spanning tree of G. That is, there is a minimum spanning tree of G that does not include e.

(d) Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree.

(e) Given a graph G and a minimum spanning tree T , suppose that we decrease the weight of one of the edges in T . Show that T is still a minimum spanning tree for G. More formally, let T be a minimum spanning tree for G with edge weights given by weight function w. Choose one edge $(x, y) \epsilon T$ and a positive number k, and define the weight function w' by

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y) \\ w(x, y) - k & \text{if } (u, v) = (x, y) \end{cases}$$

Show that T is a minimum spanning tree for G with edge weights given by w'.

(f) Given a graph G and a minimum spanning tree T , suppose that we decrease the weight of one of the edges not in T . Give an algorithm for finding the minimum spanning tree in the modified graph.

(g) Kruskals algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskals algorithm so that the algorithm returns T.

4. Recall the problem of making change for a non-negative integer amount, a, given coin denominations $\langle c_0, \ldots, c_{k-1}\rangle$, where each c_j is a positive integer, $c_j < c_{j+1}$ for all $j = 0, \ldots, k-2$ and $c_0 = 1$.
   We observed that the problem possesses optimal substructure as expressed by the following recurrence for $m[i]$, the minimum number of coins we need to make up amount i. Also, following is pseudocode for an algorithm based on dynamic programming that outputs $m[a]$

$$m[i] = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } i < 0 \\ 1 + \min\limits_{0 \le j \le k-1} \{m[i - c_j]\} & \text{otherwise} \end{cases}$$

```
M(i, ⟨c_0, ..., c_{k-1}⟩)
// Non Memoised version
if i = 0 then return 0
if i < 0 then error
ret ← i
foreach j from 0 to k − 1
      if c_j < i
            x ← 1 + M(i − c_j, ⟨c_0, ..., c_{k-1}⟩)
            if ret > x then ret ← x
return ret
```

```
M-DP(i, ⟨c_0, ..., c_{k-1}⟩)
// Memoised version, Bottom-up
if a < 0 then error
m ← new array [0, ... a]
m[0] ← 0
foreach i from 1 to a
      m[i] ← i
      foreach j from 0 to k − 1
            if c_j < i
                  if m[i] > 1 + m[i − c_j]
                        m[i] = 1 + m[i − c_j]
return m[a]
```

Part I

(a) Write down another recurrence, this one for $S[a]$, that computes the the number of combinations that make up that amount a. *[Assume that you have infinite number of each kind of coin, and that the order of coins doesnt matter]*

(b) Write down pseudo-code for a new version of the algorithm based on dynamic programming that outputs $S[a]$, the number of combinations. Can you give both Bottom-up and Top-Down approaches?

(c) Write down pseudo-code for a new version of the algorithm based on dynamic programming that outputs $m[a]$ but this time using the top down approach

(d) Write down pseudo-code for a new version of the algorithm based on dynamic programming that outputs $m[a]$ along with the coin set.

Part II

Humans unlike algorithms aren't always the most efficient. Given the coin denominations $\langle 1, 12, 19\rangle$ for making a sum of 24 we would often choose to return $\langle 19, 1, 1, 1, 1\rangle$ instead of the most efficient $\langle 12, 12\rangle$.

(a) What algorithmic design approach are we used to using in our day to day life for computing change given that we have coins of denomination $\langle 1, 2, 5, 10 \rangle$?
(b) Does this approach work for combinations with other denominations? What is the alternative algorithm approach that we should use instead?