SID: 510459613

Problem 1

a)

line 2: initialising a result value = 0 takes O(1) time
line 3: loops through all values inside A, which is n elements perfoming the actions within the loop all take O(1) time
inside that loop O(1) time if statement
add the element to the result O(1) time

returning the result is O(1) time

total time = O(1) + O(n)
total time = O(n)

b)
$T(n) = T(n/2) + O(1)$

$a = 1$
$b = 2$
$f(n) = O(1)$

first calculate $n^{\log_b a}$ to find if it is > than f(n) = to f(n) or < than f(n), and thus find the case
$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

MASTER THEORUM

Case 1: $f(n) < n^{\log_b a}$ AKA $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$
$T(n) = \Theta(n^{\log_b a})$

Case 2: $f(n) = n^{\log_b a} \cdot \log^k n$ AKA $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ for $k \geq 0$
$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1}(n))$

Case 3: $f(n) > n^{\log_b a}$ AKA $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $a \cdot f(n/b) \leq \delta f(n)$ for $\epsilon > 0$ and $\delta < 1$ then
$T(n) = \Theta(f(n))$

We have case 2 if $k = 0$

which means:

$T(n) = O(n^{\log_b a} \cdot \log^{k+1}(n))$
$T(n) = O(1 \cdot \log^{0+1}(n))$
$T(n) = O(\log^1(n))$
$T(n) = O(\log(n))$

c)
this is an attempt at a greedy algorithm. that allways picks the greatest value rather than taking the actual size of the game into account

given a set of (fi, si) fi = the fun of the game and si = the space the game occupies.

Say the size of the shelf is 10
f = {10, 9, 8}
s = {10, 5, 5}

the algorithm would take the value 10. and would have an overall fun of 10.

Where the actual greatest fun value would be 9 + 8, 17.

Problem 2

a) and b)

{(D, E), (C,F), (E,F), (C,B), (A, D)}

1. (D, E)
2. (C,F)
3. (E,F)
4. (C,B)
5. (A, D)

Problem 3

a)
Create: simply allocate space for 1 element, and allocate space inside that. both operations take O(1)

Get: would also simply access the valye stored at a[i][j]

the array would keep track of the size it needs to be in, i.e. n wide and n long
it would also keep track of the actual size of the array length

increase size:
if the n value is greater than the actual size of the array, then square the array

length

when its doubled you need to fill in the values from the old row

when you get to the last row create a new row at the bottom

b)
because it increases the size everytime it is full the data structure ensures there is always space for the new n + 1 values.

c)
To create a new one, it would initialise 2 arrays, one within the other, which would take O(1) time for the first O(1) time for the second, which results in O(1) time

Accessing variables within an array takes O(1) time. (simply pointer addition) accessing an array within an array would also take O(1) time. in total that is O(1) time

creating a larger row of size n + 1, would take O(1) time, this is done O(n + 1) times (for the new height of the dynamic matrix). if the total size of the matrix is n, then looping through and moving every element from the original matrix to the new one would take n. (because you are going through every element in the matrix)

total time = O(n + 1) + O(n)

Problem 4

a)
$T \cap S$ are the edges they have in common.
S/T are the edges in S that are not in T

T and S will have the same number of edges, if it is a minimum spanning tree they will both have the MINIMUM edges avaliable
take the T/S to see which edges in T are not in S. the differing edges in T but not in S.

then you can add an edge from S/T into T.
then take an edge from T/S and remove it from T, that the new edge was connected to
continue doing this until you get S

common_edges = take the common values of S and T

```
def swapping_trees(S, T):
        differing_S = take the S/T
        differing_T = take the T/S
        swapping_sequence = empty list {}

        add S to the swapping sequence
        for every value in differing_S:
                //swap the edges
                s_edge = differing_S.pop()
                t_edge = differing_T.pop()

                R = T (make a copy of T and save that as R)

                add s_edge to R
                remove t_edge from R (if the edge was part of a cycle)
                add R to swapping_sequence

        return swapping_sequence
```

b)

- for the algorithm to be correct, every value inside swapping sequence must be a tree
  - this can be proven by defining a tree then induction
  - by adding an edge, it means that the tree is no longer a minimum spanning tree, and because every vertice is connected, that means that there must be another route to a vertex that has already been connected in the tree. This breaks the cycle property that minimum spanning trees have, meaning we are able to remove an edge, that is part of a newly created cycle.
  - To save time, you remove the edge that isnt a part of S/T, this restores the cycle and cut edge property to the Rth tree
  - By doing this every time you ensure a minimum spanning tree in each value of R
- value Ri and Ri+1 must be one swap away from each other
  - swapping is to remove and add a new value to the trees. This is done in every iteration
- the value at the beginning must be T, and the end result of swapping sequence must be S

- if all edges from S/T are added
- and everything from T/S is removed
- what you have left is S

c) Analyze the time complexity of your algorithm.

taking the S/T would take O(n^2) time to loop through all the edges of S
and check if they are in T
similarly, taking T/S would also take O(n^2) time
initialising the swapping sequence value would take O(1) time
adding S to the swapping sequence is also O(1) time

looping through the differing values could potentially take O(n) time is all values in S are not in T.
poping takes O(1)
poping takes O(1)
making a deep copy would take O(n)
adding an edge to R would take O(1)
removing an edge would need to loop through and take it out of R taking O(n)
adding r would be O(1)

total time = O(n^2) + O(n^2) + O(n) *(O(n)) + O(n) O(1) + O(1))*
total time = 2O(n^2) + O(n) (2O(n))
total time = 2O(n^2) + 2O(n^2)
total time = 4O(n^2)
total time = O(n^2)