DAA - 2017

1.

a) The defining features of divide and conquer are that the problem is made of several smaller problems ~~[illegible]~~. We make the problems smaller and smaller until they are sufficiently small (base case), where can solve them by brute force. Then we need to "glue" the subproblems together to compose a solution for a bigger problem. We proceed until we reach the desired result. The subproblems are much smaller and don't overlap. Suppose we have x and y (both n-bit).

x $\underline{\quad x_L \quad | \quad x_R \quad}$ $\quad x = x_L \cdot 2^{\frac{n}{2}} + x_R$

y $\underline{\quad y_L \quad | \quad y_R \quad}$ $\quad y = y_L \cdot 2^{\frac{n}{2}} + y_R$

We want $(x_L \cdot 2^{\frac{n}{2}} + x_R)(y_L \cdot 2^{\frac{n}{2}} + y_R) =$

$= x_L y_L 2^n + x_L y_R \cdot 2^{\frac{n}{2}} + x_R y_L \cdot 2^{\frac{n}{2}} + x_R y_R$

These bits can be calculated by multiplying $x_L y_L$, $x_R y_R$ and $(x_L + x_R)(y_L + y_R)$. The subproblems are multiplying two numbers with $\frac{n}{2}$ bits =>

$T(n) = 3T(\frac{n}{2}) + O(n)$

By the Master Theorem => $T(n) = O(n^{\log_2 3})$

b)
```
def search(arr: Array[Int], l: Int, r: Int): Int = {
    if (r - l > 1) {
        val m = l + (r - l)/2
        if (arr(l) < arr(m)) search(arr, m+1, r) else
                             search(arr, l, m)
    }
    return l
```

c) Master Theorem: If $T(n)$ is the number of steps to do an algorithm ~~of step~~ of size n, and we know that:

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$$

time to combine the solutions

$$\Rightarrow T(n) = O(n^d) \text{ if } d > \log_b a$$
$$T(n) = O(n^d \log_b n) \text{ if } d = \log_b a$$
$$T(n) = O(n^{\log_b a}) \text{ if } d < \log_b a$$

$$\Rightarrow I: T(n) = 3T\left(\left\lceil \frac{n}{\frac{3}{2}} \right\rceil\right) + O(1)$$

$$T(n) = n^{\log_{\frac{3}{2}} 3}$$

$$O(n^2)$$

$$II: T(n) = 9T\left(\left\lceil \frac{n}{3} \right\rceil\right) + O(n) \Rightarrow 9T\left(\left\lceil \frac{n}{3} \right\rceil\right) + O(n^2) + O(n)$$

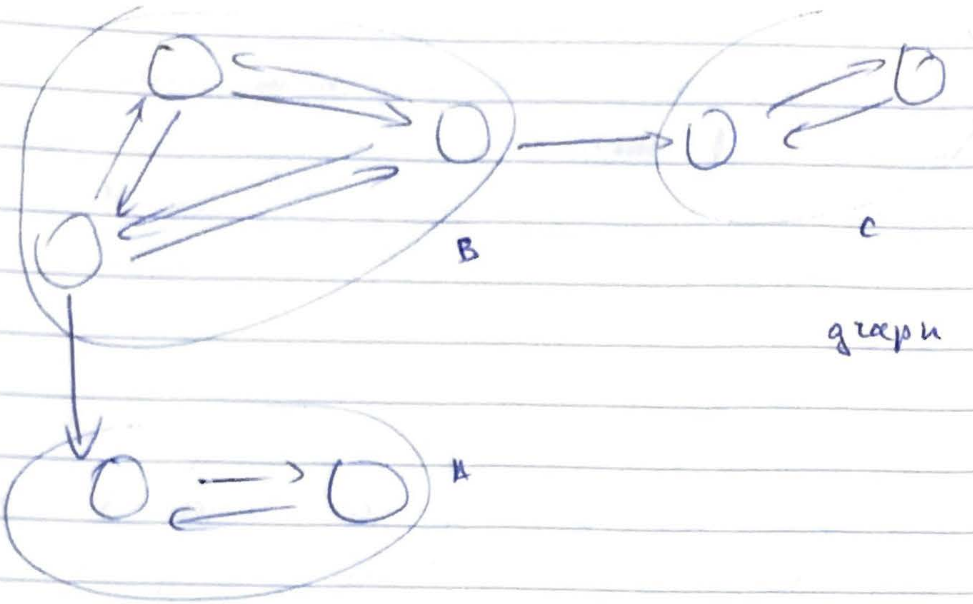
$$III: T(n) = 3T(n-1) + O(1) = c + 3c + 9c + \dots + 3^{n-1}c =$$
$$= c \cdot \frac{3^{n-1} - 1}{3 - 1} = c \cdot \frac{3^{n-1} - 1}{2}$$

$$\Rightarrow T(n) = O(3^n)$$

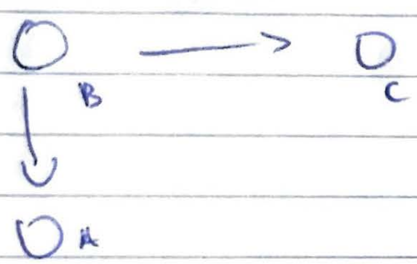
III is the slowest, because it is exponential time, whereas the others are polynomial. I is the fastest, because it is $O(n^{\log_{\frac{3}{2}} 3})$

2.

a) SCC of a graph is the biggest set of vertices, so that every pair of vertices in it are strongly connected. v and u are strongly connected iff there is a path from v to u, and there is a path from u to v. SCC graph is such that every SCC is represented by one node, and the only edges in that graph are the edges between the SCCs.



graph



SCC graph

b) Run DFS to compute the ~~discovery~~ finishing times. Start with the vertex with the highest finishing time. Compute G^T. Run DFS from that vertex in G^T.
Worst case running time is $O(|V| + |E|)$. DFS is linear, computing G^T is linear and the last DFS is also linear.

c) Suppose u, v are in different SCCs. But if the are infinitely many times in p, then there must be a path $u \xrightarrow{p_1} v \xrightarrow{p_2} u$ => u and v are strongly connected => they must be in 1 SCC. Suppose they are not in SCC => they are strongly connected => ↯.

d) If there is an infinite path from s, then s must be connected to an SCC. So we need to find out, whether s ~~[illegible]~~ is connected to an SCC. BFS to "shrink" V only to the nodes that ~~[illegible]~~ s is connected to. Then run DFS to compute finishing times. Compute G^T. Compute SCC. ~~If~~ there is an SCC, then there is an infinite path. The worst-case running time is $O(|V| + |E|)$.

7.

a) A spanning tree is a path, which goes through all the nodes exactly ~~once~~ once, and goes through each edge at most once. A MST of G is the spanning tree, whose sum of weights of the edges in it is the lowest among all trees.

b) Kruskal's algorithm is taking the ~~___~~ edge with smallest weight, which does not create a cycle.

```
kruskal(V, E){
    A = null
    Q = make - min Queue(E)
    S = make _ set(V)
    while(S.size != 1)
        e = Q.pop
        if(e.connect)   // e.connect is true if it connects two sets
            A = A ∪ e
    return A
}
```

c) Kruskal's algorithm is considered greedy, because at each step it takes the ~~___~~ edge with the minimal weight.

d) ~~___~~ $A = \{(d,f); (f,c); (c,a); (h,e); (h,f); (e,b); (e,g)\}$

e)

e) In the beginning $S = \{a, b, c, d, e, f, g, h\}$

After 1st iteration:

$A = \{(a, c)\}$

$S = \{\{a, c\}, b, d, e, f, g, h\}$

After 2nd,

$A = \{(a, c), (b, e)\}$

$S = \{d, f, g, h, \{a, c\}, \{b, e\}\}$

After 3rd,

$A = \{(a, c), (b, e), (d, f)\}$

$S = \{g, h, \{a, c\}, \{b, e\}, \{d, f\}\}$

After 4th:

$A = \{(a, c), (b, e), (d, f), (g, e)\}$

$S = \{h, \{a, c\}, \{b, e, g\}, \{d, f\}\}$

After 5th,

$A = \{(a, c), (b, e), (d, f), (g, e), (h, e)\}$

$S = \{\{a, c\}, \{b, e, g, h\}, \{d, f\}\}$

After 6th:

$A = \{(a, c), (b, e), (d, f), (g, e), (h, e), (c, f)\}$

$S = \{\{b, e, g, h\}, \{d, f, a, c\}\}$

After 7th,

$A = \{(a, c), (b, e), (d, f), (g, e), (h, e), (c, f), (h, f)\}$

$S = \{\{a, b, c, d, e, f, g, h\}\}$

I am going to prove it by Induction. Suppose A is already part of an MST. The Cut lemma is: suppose $S, V \setminus S$ is a cut, and A respects that cut. Then the light edge with A is a part of an MST. At line 4 we examine a path, which is a subset of an MST, If S ~~A~~ nodes of A and $V \setminus S$ are the rest of the ~~edges~~ nodes, then at line 5 we choose the light edge. So $A \cup \{\text{light edge}\}$ is part of an MST.

The base case for the Induction is $A = \emptyset$, which is a subset of all trees.

8.

a) Dynamic programming is a where we combine the optimal substructures of smaller problems to do an optimal solution to the problem we have. The subproblems are ordered from smallest to largest. Principle of optimality is when we can construct a solution to a bigger problem from optimal solutions of smaller ones. We can use DP for the alignment problem, because there is an optimal substructure.

b) Suppose $s(i,j)$ contains ~~all~~ the maximum score of all possible alignments of $x[0..i]$ and $y[0..j]$. The optimal substructure is:

$$s(i,j) = \max\{s(i-1,j) - g \,;\, s(i,j-1) - g \,;\, s(i-1,j-1) + 1\}$$

~~$s(i-1,j) - g$ is the case when $x(i) \neq y(j)$~~
~~$s(i,j-1) - g$ is the case when $x(i) \neq y(j)$~~

$s(i-1,j) - g$ is the case when $\frac{\square}{y(j)}$

$s(i,j-1) - g$ is when $\frac{x(i)}{\square}$

$s(i-1,j-1)$ is when $\frac{x(i)}{y(j)}$

Initialization: $s(0,j) = -gj$
$s(i,0) = -gi$

Worst-case running time is $O(mn)$.

c) To output the optimal alignment, we can have two ~~[illegible]~~ strings. When we have calculated the maximum score, we can retrace to see which step we took. We can have a list, where each element has data 0, 1 or 2.

[redacted]

0 - characters are the same
1 - inserted space in string 1
2 - Inserted space in string 2
Retrace the list and build the strings accordingly.

d) ~~Iterate for i from 0 to m, iterate j from 0 to n~~
Suppose the length of x' is m' and length of y' is n'. Apply the algorithm to it. We have $O(m^2)$ choices for m', and $O(n^2)$ choices to n' => upper bound on the running time is $O(m^3 n^3)$.

e) (Ran out of time)