DESIGN AND ANALYSIS OF ALGORITHMS

GABRIEL MOISE

HT 2019

PROBLEM SHEET 3

Dynamic Programming

* **Question 1**

Define

$\Delta[j]$ = maximum sum of a contiguous subsequence of S, ending at position j, $1 \le j \le n$

Then, we have

$$\Delta[j] = \max\{0, A[j] + \Delta[j-1]\}, \quad j \ge 2 \qquad (\Delta[1] = \max\{A[1], 0\})$$

($A[j]$: we add the j^{th} element to the (maximum) previous contiguous subsequence; 0: if we get negative sum, we can choose just 0)

The result will be $\max_{1 \le j \le n} \Delta[j]$ and we will seach from that position backwards until we find a position i, $0 \le i < j$ such that $\Delta[i] = 0$ (if it was > 0, then we could add the contiguous subsequence ending there, too), and the array will be $A[i+1], A[i+2], \dots, A[j]$ (we consider $A[0] = 0$)

Pseudode:

```
1. A[0] = 0
2. Δ[1] = max{A[1], 0}
3. for j=2 to n
4.     Δ[j] = max{0, A[j] + Δ[j-1]}        // O(n)
5. maxSum = 0 ; pos = 0
6. for j=1 to n
7.     if Δ[j] > maxSum
8.         maxSum = Δ[j] ; pos = j          // O(n)
9. i = pos - 1
10. while (Δ[i] > 0)  i = i - 1             // O(n)
11. left = i + 1
12. for i=left to pos                       // O(n)
13.     print A[i]
```

⇒ Total time: $O(n)$ ⇒ linear-time algorithm

Question 2

$ed(A, B)$ = the smallest number k such that A can be transformed in B in k moves, where a move is an insertion or deletion of a letter.

Let A^i be the prefix of A of length i, $0 \le i \le n$

(a) $ed(A^0, B^j) = j$ as we need j insertions to get from A^0 to B^j. Similarly, for $ed(A^i, B^0) = i$ we need i deletions to get from A^i to B^0.

(b) $i, j > 0$

If A^i and B^j have the same final letter, then we only have to edit A^{i-1} into B^{j-1}, so $ed(A^i, B^j) = ed(A^{i-1}, B^{j-1})$ in this case.

If A^i and B^j do not have the same final letter, to transform A^i into B^j we can either:

- transform A^i into B^{j-1} and insert $B[j]$ (the j^{th} character of B) at the end $\Rightarrow ed(A^i, B^{j-1}) + 1$
- delete $A[i]$ and transform A^{i-1} into $B^j \Rightarrow ed(A^{i-1}, B^j) + 1$

Therefore, we want the minimum of these two:

$$ed(A^i, B^j) = \min\{ed(A^i, B^{j-1}) + 1, ed(A^{i-1}, B^j) + 1\} = 1 + \min\{ed(A^i, B^{j-1}), ed(A^{i-1}, B^j)\}$$

(c) Pseudocode:

```
1.  for j=0 to m
2.      ed[0,j] = j      // Using (a) to set the initial subproblems
3.  for i=0 to n
4.      ed[i,0] = i
5.  for i=1 to n         // Building the matrix ed with our two formulas from (b)
6.      for j=1 to m
7.          if A[i] = B[j]
8.              ed[i,j] = ed[i-1,j-1]   // Same last letter
9.          else ed[i,j] = 1 + min{ed[i,j-1], ed[i-1,j]}   // Different last letter
10. return ed[m,m]       // The answer we want is ed(A,B) = ed(A^m, B^m) = ed[n,m]
```

Total time: $O(n \cdot m)$

(d) A = "rhymes"
B = "reason"

ed	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	2	3	4	5	6
3	3	2	3	4	5	6	7
4	4	3	4	5	6	7	8
5	5	4	3	4	5	6	7
6	6	5	4	5	4	5	6

○ = worst option (based on comparisons)
○ = best option
□ = largest number of moves

To obtain B from A we need 6 moves:
- as $A[6] \neq B[6]$, we needed 1 move + to obtain B^5 from A^6 (as 5<7), so the last operation was [insert "n"].
- as $A[6] \neq B[5]$, we needed 1 move + to obtain B^4 from A^6 (as 4<6), so the last operation was [insert "o"]
- as $A[6] = B[4]$, we needed to obtain B^3 from A^5
- as $A[5] \neq B[3]$, we needed 1 move + to obtain B^2 from A^5 (as 3<5), so the last operation was [insert "a"]
- as $A[5] = B[2]$, we needed to obtain B^1 from A^4
- as $A[4] \neq B[1]$, we needed 1 move + to obtain B^1 from A^3 (as 4>2), so the last operation was [delete "m"]
- as $A[3] \neq B[1]$, we needed 1 move + to obtain B^1 from A^2 (as 3>1), so the last operation was [delete "y"]
- as $A[2] \neq B[1]$, we needed 1 move + to obtain B^1 from A^1 (as 2>0), so the last operation was [delete "h"]
- as $A[1] = B[1]$, we needed to obtain B^0 from $A^0 \Rightarrow$ 0 moves

The sequence of moves is:

- delete "h" (A="rymes")
- delete "y" (A="rmes")
- delete "m" (A="res")
- insert "a" (A="reas")
- insert "o" (A="reaso")
- insert "n" (A="reason")

→ 6 moves

To recover the moves, we start from i=m, j=m and compare A[i] with B[j]. If they are equal, we do nothing and we decrease i and j by 1. Otherwise, we choose the minimum from c[i-1,j] and c[i,j-1] and if we have c[i-1,j] < c[i,j-1], then we deleted A[i], otherwise we inserted B[j].

From the table, ed(A^i, B^j) is the largest when i=4 and j=6, meaning for A="rhym" to B="reason".

Question 3

(a) Let $c[i] = \begin{cases} 1, & \text{if we can obtain the sum } i \\ 0, & \text{otherwise} \end{cases}$, $0 \le i \le V$

Then, we have $c[x_1] = c[x_2] = \dots = c[x_n] = 1$ and

$c[0] = 1$

$c[i] = \bigvee_{j=1}^{n} c[i-x_j]$, if $i \ge x_j$ (here we use V as "big OR" meaning that if at least one of $c[i-x_1], c[i-x_2], \dots, c[i-x_n]$ is 1, then $c[i]=1$ and $c[i]=0$, otherwise)

Pseudocode:

```
1.  c[0] = 1                    // for i=0 to V, c[i]=0 // initialising the array, O(V)
2.  for j=1 to n
3.      c[x_j] = 1              // O(n)
4.  for i=1 to V                // O(nV)
5.    for j=1 to n
6.      if i > x_j
7.        if c[i-x_j] = 1
8.          c[i] = 1
9.  if c[V] = 1
10.   print "YES"
11. else print "NO"
```

Total time: $O(nV)$

(b) If we are allowed to use each denomination at most once, we can reduce the problem to something similar to the Knapsack problem (without repetition). Let

$$\Delta[i,j] = \begin{cases} 1, & \text{if we can obtain the sum } i \text{ by using only denominations from } x_1, x_2, \dots, x_j \\ 0, & \text{otherwise} \end{cases}$$

Then, we will have

$\Delta[i,0] = 0$, for $1 \le i \le V$ (we can't obtain a positive sum without denominations)

$\Delta[0,j] = 1$, for $0 \le j \le n$ (we can always obtain the sum 0)

$\Delta[i,j] = \Delta[i-x_j, j-1]$ (if $i \ge x_j$) $\vee$ $\Delta[i,j-1]$ (we can either obtain sum i from the first $(j-1)$ denominations – $\Delta[i,j-1]$ or we can obtain sum $i-x_j$ from the first $(j-1)$ denominations and then use x_j to get i – $\Delta[i-x_j, j-1]$ (if $i \ge x_j$)

Pseudocode:

```
1.  for i = 1 to V                    // O(V)
2.      Δ[i,0] = 0
3.  for j = 0 to n                    // O(n)
4.      Δ[0,j] = 1
5.  for i = 1 to V                    // O(nV)
6.      for j = 1 to n
7.          if Δ[i,j-1] = 1
8.              Δ[i,j] = 1
9.          else if (i ≥ x_j) and (Δ[i-x_j,j-1] = 1)   // if i < x_j, the compiler ignores the other condi-
10.             Δ[i,j] = 1                              // tion, so we won't get an OUT OF BOUND INDEX error
11. if Δ[V,n] = 1
12.     print "YES"
13. else print "NO"
```

for i=0 to V
  for j=0 to n
    Δ[i,j] = 0
// initialising the matrix, O(nV)

Total time: $O(nV)$

Question 4

In order to obtain the value V from denominations $x_1, x_2, \dots, x_n$ (with $x_1 = 1$), we now have to constrain ourselves to using at most k coins. To do this, we will use the COINS array, with the property

COINS[m] = minimum number of coins needed to obtain the sum m (every sum can be obtained since $x_1 = 1$, so we can use m coins of the first type)

Then, we have

$$COINS[m] = 1 + \min\{C[m-x_i] : 1 \le i \le n, m \ge x_i\} \text{ (for } 1 \le m \le V\text{) and } COINS[0] = 0$$

To determine whether we can obtain the sum v using at most k coins, we simply compare $coins[v]$ with k.

Pseudocode: (we use the array C instead of coins here)

```
1.  for m = 0 to V
2.      C[m] = 0          // initialising the array, O(V)
3.  for m = 1 to V        // O(V · O(lines 4-8))
4.      min = +∞
5.      for i = 1 to n    // find min{C[m - x_i] : 1 ≤ i ≤ n, m ≥ x_i}, O(n)
6.          if m ≥ x_i
7.              if C[m - x_i] < min
8.                  min = C[m - x_i]
9.      C[m] = 1 + min    // Using the formula for coins[m]
10. if k ≥ C[V]
11.     print "YES"       // We need at most k coins
12. else print "No"       // We need more than k coins
```

Total time: $O(nV)$

* Question 5

To determine the longest palindromic sequence in a string, let

$L[i,j]$ = the length of the longest palindromic sequence from $\{x_i, x_{i+1}, \dots, x_j\}$. Then, for the recurrence relation we can have: (we need the maximum here)

- the longest palindromic sequence in $\{x_{i+1}, x_{i+2}, \dots, x_j\} \Rightarrow L[i+1, j]$
- the longest palindromic sequence in $\{x_i, x_{i+1}, \dots, x_{j-1}\} \Rightarrow L[i, j-1]$
- the longest palindromic sequence in $\{x_{i+1}, x_{i+2}, \dots, x_{j-1}\}$ if we add to that the characters x_i and x_j (only if they are equal) $\Rightarrow L[i+1, j-1] + 2 \cdot eq(x_i, x_j)$

We initialise $L[i+1, i] = 0$ for $1 \le i \le n-1$ (no subsequence)

$L[i, i] = 1$ for $1 \le i \le n$ (every character is a palindromic sequence)

Pseudocode:

```
1.  for i = 1 to n-1      // initialising, O(n)
2.      L[i+1, i] = 0
3.  for i = 1 to n        // initialising, O(n)
4.      L[i, i] = 1
5.  for i = n downto 1    // O(n^2)
6.      for j = i+2 to n  // We already know L[i,i] and L[i+1,i], so j ≥ i+2
```

```
7.    if x[i] = x[j]    // the eq(i,j) part
8.        eq = 1
9.    else eq = 0
10.   L[i,j] = max{L[i+1,j], L[i,j-1], L[i+1,j-1] + 2·eq}
11. return L[1,m]
```

Total time : $O(m^2)$

Question 6

Let's consider n matrices that we want to multiply:

$M_1\ [a_0 \times a_1]$, $M_2\ [a_1 \times a_2]$, $M_3\ [a_2 \times a_3]$, ..., $M_{n-1}\ [a_{n-2} \times a_{n-1}]$, $M_n\ [a_{n-1} \times a_n]$.

We then create the array $A = \{a_0, a_1, \dots, a_{n-1}, a_n\}$ on which we apply the following operations: we delete an element a_i from the array ($1 \le i \le n-1$) and we add to cost, the product $a_{i-1} * a_i * a_{i+1}$. This process is equivalent to multiplying M_i with M_{i+1}. We repeat this until we reach only two elements, which will always be a_0 and a_n. The task is to do this by generating the minimum cost possible.

Let $\Delta[i,j]$ be the minimum cost needed for the subarray $\{a_i, a_{i+1}, \dots, a_j\}$, with $j \ge i+2$, $0 \le i \le n-2$, $2 \le j \le n$. We can easily see that for $\Delta[i,i+2]$, we have the cost $a_i * a_{i+1} * a_{i+2}$.

We need to create a function CUT(i,j,k,l) which will be

CUT$(i,j,k,l) = \min\{a[i]*a[j]*a[k] + a[i]*a[k]*a[l],\ a[j]*a[k]*a[l] + a[i]*a[j]*a[l]\}$.

It is then obvious that $a[i,i+3] =$ CUT$(i, i+1, i+2, i+3)$. But, we created this function for other reasons. Going back to the general case, we imagine that the last two matrices are $M[a_i \times a_j]$ and

Question 6

Let's consider the n matrices that we want to multiply:

$M_1[a_0 \times a_1]$, $M_2[a_1 \times a_2]$, ..., $M_n[a_{n-1} \times a_n]$.

In order to compute the product $M_1 M_2 \cdot \ldots \cdot M_n$, we can have at the last step, a matrix multiplication between $M[a_0 \times a_k]$ and $N[a_k \times a_n]$, where M and N are obtained from optimal ways of bracketing the products $(M_1 M_2 \cdot \ldots \cdot M_k)$ and $(M_{k+1} M_{k+2} \cdot \ldots \cdot M_n)$, respectively, for $1 \leq k \leq n-1$. Therefore, let

$D[i,j]$ = the minimum number of operations needed to calculate $M_i M_{i+1} \cdot \ldots \cdot M_j$.

To construct this dynamic matrix, we need to start from $d=0, d=1, \ldots, d=n-1$, where $d = j-i$. We have:

$D[i,i] = 0$, for all $1 \leq i \leq n$

$D[i,i+1] = a_{i-1} * a_i * a_{i+1}$, for all $1 \leq i \leq n-1$

And ($M_i M_{i+1} \cdot \ldots \cdot M_k$: $M[(i-1) \times k]$ and $N[k \times (i+d)]$; $M_{k+1} M_{k+2} \cdot \ldots \cdot M_{i+d}$)

$$D[i,i+d] = \min \{ D[i,k] + D[k+1, i+d] + a_{i-1} * a_k * a_{i+d} : i \leq k < i+d \}, \text{ for all } 1 < d < n \text{ and } 1 \leq i \leq n-d$$

Pseudocode:

```
1.  for i = 1 to n                    // Initialising, O(n)
2.      D[i,i] = 0
3.  for i = 1 to n-1                  // Initialising, O(n)
4.      D[i,i+1] = a_{i-1} * a_i * a_{i+1}
5.  for d = 2 to n-1                  // Lines 5-9 need O(n^3) because of the 3 nested for-loops
6.      for i = 1 to n-d
7.          D[i, i+d] = +∞
8.          for k = i to i+d-1
9.              D[i,i+d] = min{ D[i,i+d], D[i+k] + D[k+1, i+d] + a_{i-1} * a_k * a_{i+d} }
10. return D[1, n]
```

Total time: $O(n^3)$ $\Rightarrow$ this problem can be solved in time polynomial in n.

Graphs: Paths and Cycles

Question 7

(a) Let $G=(V,E)$ be an undirected graph with $V=\{1,2,\dots,n\}$ and no self-loops.

Then $(i,i)\notin E$ $(\forall)\, i\in V$ and $(\forall)\, i\neq j$ we have $(i,j)\in E \iff (j,i)\in E$. Then, we only focus on (i,j) with $i<j$; $i,j\in V$

$i=1 \Rightarrow$ at most $n-1$ edges $(2,3,\dots,n)$

$i=2 \Rightarrow$ at most $n-2$ edges $(3,4,\dots,n)$

$\vdots$

$i=n-1 \Rightarrow$ at most 1 edge(n)

$i=n \Rightarrow$ no edge

$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \quad (+)$$

$\frac{(n-1)\,n}{2}$ edges.

Now, each edge can be in E or not, so we have $2^{\frac{(n-1)\,n}{2}}$ possible E sets $\Rightarrow$ $\boxed{2^{\frac{(n-1)\,n}{2}} \text{ graphs}}$

(b) Let $a_1, a_2, \dots, a_n$ be the nodes of the graph. The graph has k connected components, so we can say that these are $C_1, C_2, \dots, C_k$. Therefore, we know that

$a_{i_1}\in C_1$, $a_{i_2}\in C_2, \dots, a_{i_k}\in C_k$, for $i_1, i_2, \dots, i_k \in \{1,2,\dots,n\}$ distinct.

Now, the remaining $n-k$ nodes from the graph need to be in on component, so there must be an edge between them and one node from a specific component (there can be edges to other nodes from that component, too, but we are only interested in minimizing the number of edges). Then, we have at least $\boxed{n-k \text{ edges}}$.

(c) Let $G=(V,E)$, with $V=\{a_1, a_2, \dots, a_n\}$ undirected connected graph. If we connect each node from the graph to a_1, we have a connected graph with $\boxed{n-1 \text{ edges}}$. Let's suppose that we have fewer edges. We start by connecting two nodes (a_i and a_j) with an edge. We will colour each node that has a path to a_i with red. So far, two red nodes. At each step, we can colour at most one node in red (if we want to connect a_k and a_l, two different nodes, and the connect one of them to a_i, we colour 2, but this doesn't contradict the statement as if we reverse the order, we colour first a_k and the a_l). So, after $(n-3)$ steps ($|E|\leq n-2$, so we are only left with $n-3$ steps after connecting a_i with a_j), we are left with $(n-1)$ connected nodes, so the graph is not connected.

(d) Let $G=(V,E)$ be an undirected acyclic graph. Then, if we say that $C_1, C_2, \dots, C_k$ are its connected components, with $1 \le k \le |V| = n$, we have that no connected component has a cycle.

The maximum number of edges that an acyclic connected component C_i can have is $|C_i|-1$. Let's suppose that we can have $|C_i|$ edges. Then, by the Pigeonhole principle, one vertex has to have at least two neighbours. Let a_1 be that node and a_2 and a_3 be the neighbours. Now, each vertex a_j with $4 \le j \le |C_i|$ needs to be connected to this group of three. However, it can only have an edge to one of the current group's vertices, otherwise it would form a cycle. So, we have $(|C_i| - 4 + 1) + 2 = |C_i| - 1$ edges.

So, the total number of edges is $\sum_{i=1}^{k} |C_i| - 1 = n - k$, as $|C_1| + |C_2| + \dots + |C_k| = n$. Therefore, the maximum number of edges is $\boxed{n-1}$, when the graph is connected.

*Question 8

(a) A tree has $E = \{ (\Pi[u], u) \mid u \in \underbrace{\{v_1, v_2, \dots, v_n\}}_{V} \}$, where V is the set of vertices from the tree. So, the number of edges equals the number of predecessor, which is always $n-1$.

(b) $t_n = n-1$

If the graph is not connected, then let $k \ge 2$ be the number of connected components it has. From Q7 (c), we know that each component has e_i edges at least, where $e_i = |C_i| - 1$, C_i being the set of nodes of the i^{th} component. Then $\sum_i e_i = n - k \ge n-1 \Rightarrow k = 1$ and the whole graph is connected. Because it is acyclic and undirected, it must also be a tree.

(c) For any u and v in the tree there must be a path between them and since a tree is acyclic, the path must be simple. By having two different paths from u and v, they will form a cycle starting from where they split and ending at where they meet again, so the graph will contain a cycle, which is not allowed. Therefore, the path must be unique.

(d) By adding an edge from u to v and since there already exists a path from u to v, we form a cycle starting from u. Now, by supposing that there are two (at least) cycles starting from u, they both share the edge e we added as before adding e in the tree there was no cycle. Now, the two cycles have a point from which they split u' and a point in which they meet again, v'. If e is before u' or after v', then we can reach v' in two different ways, so there is a cycle there. If e is in the path of u' and v', then the cycle which does not contain e was a cycle in the tree, which is impossible. So, the cycle is unique.

(e) Suppose u and v are vertices in G. By deleting the edge e from G we obtain a new graph G'. Since G is connected, there is a path from u to v in G. If e does not belong to that path, then u and v are connected in G', too. If it does, then in order to reach v from u we can replace e by the path that is formed after e was removed from the cycle.

Question 9

G is a finite, connected, undirected graph without loops.

- degree of i = number of edges incident on i
- circuit = nonempty finit path where target of the last edge = source of the first edge and no edge occurs twice (a vertex can occur more than once in a circuit)

(a) Let $G = (V, E)$, $V = \{a_1, a_2, \dots, a_n\}$

$\text{degree}(a_i) \geq 2 \quad (\forall)\ i \in \{1, 2, \dots, n\}$

Let's suppose that we start from a_{i_1} and we go along edges (without using an edge twice) until we can no longer continue (i.e. there are no edges left to continue). Let's assume that we have:

$a_{i_1} \to a_{i_2} \to a_{i_3} \to \dots \to a_{i_k}$, which is not a circuit, and does not contain a circuit, so $i_1, i_2, \dots, i_k$ are all distinct. If a_{i_k} has degree ≥ 2, then there must be an edge which was not used, as we only used $a_{i_{k-1}} \to a_{i_k}$, so we can't stop there, thus reaching a contradiction (there must be a circuit contained there, because there must be an $a_{i_j} = a_{i_k}$ with $j \in \{1, 2, \dots, k-1\}$ for us to use all possible edges for a_{i_k}).

(b) · Eulerian circuit = traverses every edge exactly once

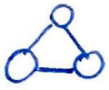
(i) We know that G has an Eulerian circuit, therefore we can form the circuit:

$a_{i_1} \to a_{i_2} \to a_{i_3} \to \dots \to a_{i_k} \to a_{i_1}$, where nodes can be visited more than once. We only focus on the number of edges incident on a specific node a_{i_j}. Every time we visit a_{i_j} in the circuit we add two to the number of edges incident on a_{i_j} that we have discovered so far $\Rightarrow$ $\text{degree}(a_{i_j}) = 2 \cdot$ #number of times it has been visited (the start node is not visited again when we finish, for that we already counted the returning edge when we started) $\Rightarrow$ every node has an even degree.

(ii) We now start by knowing that every vertex of G has an even degree.

By using (a), we know that G has a circuit.

Now, we will proceed by strong induction on $n = |V|$, with $n \geq 3$ (from there we can reason about G)

$P(3)$:  obviously has an Eulerian circuit (only possibility for G)

Now, we assume that every G with $|V| \leq n$ and even degree for every vertex has an Eulerian circuit and we'll prove that for G (with even degrees for vertices) and with $|V| = n+1$ this also is true.

As we said, we know from (a) that G has a circuit. If we remove that circuit from G, we are left with k connected components, all of size $\leq n$, so, by using the inductive hypothesis, they all have Eulerian circuit.

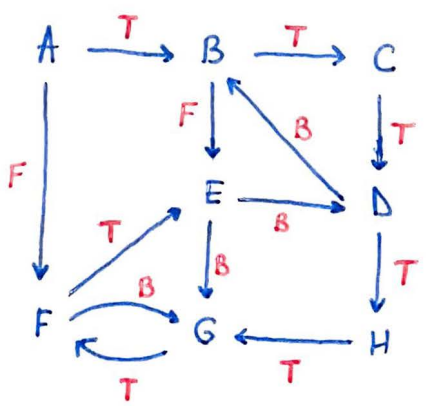
By putting the circuit $(a_1 \to a_2 \to \dots \to a_k \to a_1)$ back into the graph, we can do the following: start from a_1 and if it belongs to a connected component we go through its Eulerian circuit and come back to a_1 (this happens if $\text{degree}(a_1) \geq 2$), ~~we can't leave from the nodes from the circuit in the same connected component, otherwise the degree would be odd~~, and proceed in the same way for $a_2, a_3, \dots, a_k$ until we go through all the circuit, therefore proving that G has an Eulerian circuit, too, as we passed through all the nodes from the circuit and from the connected components.

Therefore, the induction on n is proven.

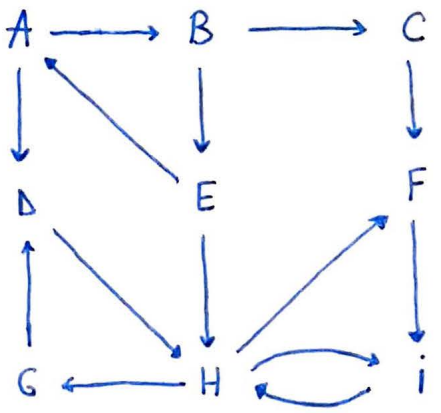
Depth-First Search and Connected Components

Question 10

	A	B	C	D	E	F	G	H
d	1	2	3	4	8	7	6	5
f	16	15	14	13	9	10	11	12
π	NIL	A	B	C	F	G	H	D



* Question 11



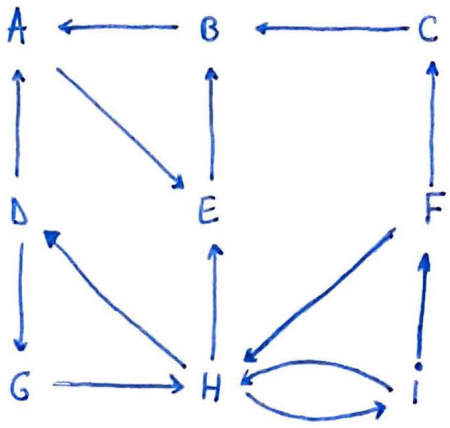
(a) • Calling DFS(G), we get

	A	B	C	D	E	F	G	H	i
d	1	2	3	8	15	4	7	6	5
f	18	17	14	9	16	13	10	11	12
π	NIL	A	B	G	B	C	H	i	F

• Compute G^T

• Call DFS(G^T) in the following order:

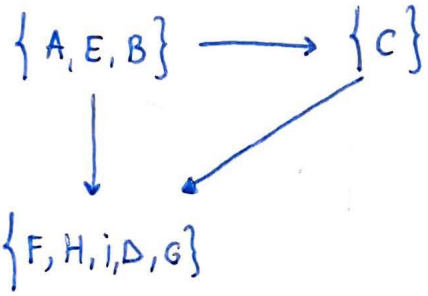
A, B, E, C, F, i, H, G, D



	A	B	C	D	E	F	G	H	i
d	1	3	7	13	2	9	14	10	11
f	6	4	8	16	5	18	15	17	12
π	NIL	E	NIL	H	A	NIL	D	F	H

Therefore, the SCCs found are (in order): $\{A, E, B\}$; $\{C\}$; $\{F, H, i, D, G\}$

(b)



(c) To make the graph strongly connected, we need to add an edge from $\{F, H, i, D, G\}$ to $\{A, E, B\}$, for example (F, B).

Question 12

G directed graph $\xrightarrow{DFS}$ F is the DFS forest produced

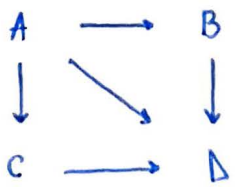
(a) If there is a path between u and v, then this path is formed of $u, q_1, \dots, q_k, v$, where $u = \pi[q_1]$, $q_1 = \pi[q_2]$, $\dots$, $q_k = \pi[v] \Rightarrow d[u] < d[q_1] < \dots < d[q_k] < d[v] \Rightarrow d[u] < d[v]$, so this is TRUE.

(b) In the example from the lecture, at page 17/32 we can see that $d[A] < d[D]$ and $(A, D) \in E$, but $(A, D) \in E_\pi$, as (A, D) is a FORWARD edge.

In general, if we have a forward edge, that it is not going to appear in the forrest, however we still have $d[u] < d[v] < f[v] < f[u]$, for (u, v) = forward edge, similar to a tree edge. So, this statement is FALSE.

(c) If G is strongly connected and we start the DFS from a node N, we know that we will reach every other node because there is a path to every other node. Thus, after the DFS, the forest which will be formed will be a tree, so this statement is TRUE.

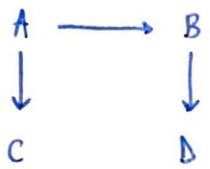
(d) Let the graph G:



then by calling DFS(G) we get:

	A	B	C	D
d	1	2	6	3
f	8	5	7	4
π	NIL	A	A	B

then the DFS forest is



which is a tree, but initially G is not a strongly connected graph, as we cannot find a path from D to A, for example. Therefore, this statement is FALSE.

* Question 13

We'll modify the DFS for an undirected graph so that we can use it to find the connected components of G and we'll show that the DFS forest contains as many trees as the number of connected components of G.

We will create an array C[v], which will contain a number between 1 and k, where k is the number of connected components of G, so we basically assign each CC with a number and all the nodes from each CC are labelled with that number.

```
DFS (V, E)
1. for u ∈ V
2.     colour [u] = WHITE
3. time = 0
4. count = 0    // the current number of CC found so far, which is also the number of trees found so far
5. for u ∈ V
6.     if colour [u] = WHITE
7.         count = count + 1    // we have found a new CC / tree
8.         DFS-VISIT (u)
```

DFS-VISIT(u)

1. time = time + 1
2. d[u] = time
3. colour[u] = GREY
4. C[u] = count // As u is in the current CC, it has to be labelled with the current value of count
5. for v ∈ Adj[u]
6. if colour[v] = WHITE
7. DFS-VISIT(v)
8. time = time + 1
9. f[u] = time
10. colour[u] = BLACK

At the end, we can see that each time we find a new CC, we are in the for loop from DFS(V,E) and therefore, we have also found a new tree in the DFS forest.

To conclude, at the end we have:

- the array C which can be easily used to identify all the connected components of G: u, v ∈ CC iff C[u] = C[v].
- count = the current number of trees found so far throughout the program, so, at the end, we get that k = total number of CCs in G = total number of trees in the DFS forest.