1. TextAlignment (see the webnotes) can be done in a forward manner. Let \( l[1], \ldots, l[n], L \), penalty function \( 0 = P[0], \ldots, P[L-1] \) be given as before. Now set \( FBAD[i] \) (\( F \) for front) as the maximal total badness for the text \( l[1], \ldots, l[i] \). Assume (to avoid the easy case) that \( l[1], \ldots, l[i] \) do not fit on one line. Assume (important!) that \( FBAD[j] \) are already known for all \( j < i \). Give a formula for \( FBAD[i] \) as the minimum of some things. From the formula, create an algorithm to determine \( FBAD[i] \) which takes time \( O(L) \). (Idea; Consider the last line when \( l[1], \ldots, l[i] \) is parsed.)

**Solution:** \( FBAD[i] \) is the maximum over all \( k \) from \( i \) down for which \( l[k], \ldots, l[i] \) fit on one line of \( FBAD[k-1] + P[GAP] \) where \( GAP = L - (l[k] + \ldots + l[i] + i - k) \). The idea is that with \( l[k], \ldots, l[i] \) as the last line the remaining text is \( l[1], \ldots, l[k-1] \) which has minimal total badness \( FBAD[i] \).

2. Consider the undirected graph with vertices 1, 2, 3, 4, 5 and adjacency lists (arrows omitted) 1 : 25, 2 : 1534, 3 : 24, 4 : 253, 5 : 412. Show the \( d \) and \( \pi \) values that result from running BFS, using 3 as a source. Nice picture, please!

**Solution:**
- BFS: 3, 2, 4, 1, 5
- \( d[3] = 0, \pi[3] = nil \)
- \( d[2] = 1, \pi[2] = 3 \)
- \( d[4] = 1, \pi[4] = 3 \)
- \( d[1] = 2, \pi[1] = 2 \)

3. Show the \( d \) and \( \pi \) values that result from running BFS on the undirected graph of Figure A, using vertex \( u \) as the source.

**Solution:**
- \( d[U] = 0, \pi[U] = nil \)
- \( d[T] = 1, \pi[T] = U \)
- \( d[X] = 1, \pi[X] = U \)
- \( d[Y] = 1, \pi[Y] = U \)
- \( d[W] = 2, \pi[W] = T \)
- \( d[S] = 3, \pi[S] = W \)
- \( d[R] = 4, \pi[R] = S \)
- \( d[V] = 5, \pi[V] = R \)
4. We are given a set $V$ of wrestlers. Between any two pairs of wrestlers there may or may not be a rivalry. Assume the rivalries form a graph $G$ which is given by an adjacency list representation, that is, $\text{Adj}[v]$ is a list of the rivals of $v$. Let $n$ be the number of wrestlers (or nodes) and $r$ the number of rivalries (or edges). Give a $O(n + r)$ time algorithm that determines whether it is possible to designate some of wrestlers as GOOD and the others as BAD such that each rivalry is between a GOOD wrestler and a BAD wrestler. If it is possible to perform such a designation your algorithm should produce it.

Here is the approach: Create a new field $\text{TYPE}[v]$ with the values GOOD and BAD. Assume that the wrestlers are in a list $L$ so that you can program: For all $v \in L$. The idea will be to apply $\text{BFS}[v]$ – when you hit a new vertex its value will be determined. A cautionary note: $\text{BFS}[v]$ might not hit all the vertices so, just like we had DFS and DFS-VISIT you should have an overall $\text{BFS-MASTER}$ (that will run through the list $L$) and, when appropriate, call $\text{BFS}[v]$.

Note: The cognescenti will recognize that we are determining if a graph is bipartite!

Solution: The idea here is to call the first wrestler GOOD. When someone is adjacent to someone GOOD they are called BAD and if they are adjacent to someone BAD they are called GOOD. But if in the adjacency list you come upon someone who has already been labelled (that is, not white) then you must check if there is a contradiction. A further problem: $\text{BFS}[v]$ will only explore the connected component of $v$, if that is labelled with no contradiction then you must go on to the other vertices. So we start with everything white. The “outside” program is:

For all $v \in L$
If $\text{COLOR}[v] = \text{WHITE}$ (*else skip*) then $\text{BFSPLUS}[v]$.

$\text{BFSPLUS}[v]$ starts by setting $\text{TYPE}[v] = \text{GOOD}$. Then it runs $\text{BFS}[v]$ with two additions. When $u \in \text{Adj}[w]$ and $u$ is white you define $\text{TYPE}[u]$ to be the opposite of $\text{TYPE}[w]$. When $u$ is not white you check if $\text{TYPE}[w] = \text{TYPE}[u]$. If not, ignore. But if so exit the entire program with NO DESIGNATION POSSIBLE printout.

5. Show how DFS works on Figure B. All lists are alphabetical, except that we put R before Q so it is the first letter. Show the discovery and finishing time for each vertex.

Solution:
Discovery order: RUYQSVWTXZ
Finishing order: WVSZXTQYUR
Stack: push(R) push(U) push(Y) push(Q) push(S) push(V) push(W) pop(W) pop(V) pop(S) push(T) push(X) push(Z) pop(Z) pop(X) pop(T) pop(Q) pop(Y) pop(U) pop(R)

6. Show the ordering of the vertices produced by \textsc{Top-Sort} when it is run on Figure C, with all lists alphabetical.

Solution: We apply \textsc{DFS} to the graph. The first letter is \textit{M} so we apply \textsc{DFS-Visit(\textit{M})}

<table>
<thead>
<tr>
<th>v</th>
<th>s[v]</th>
<th>f[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Q</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>R</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>U</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Y</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>V</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>W</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>Z</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>X</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

Note, for example, that though \textit{X} is in \textit{Adj[M]} it doesn’t affect \textsc{DFS}. At time 19 \textit{R} finishes and returns control to \textit{M}. \textit{M} looks at \textit{X} in its adjacency list but it is no longer white and so ignores it. At this stage all vertices are black except \textit{N, O, P, S} which as white. In this particular example \textit{N} is the letter right after \textit{M} but in the general case \textsc{DFS} would skip over those vertices which weren’t white. Indeed, right after \textsc{DFS-Visit} all vertices are white or black. So next we do \textsc{DFS-Visit(N)}. Note that the time does \textit{not} restart! Note also that the now black vertices, such as \textit{U} \in Adj(\textit{N}) and \textit{R} \in Adj(\textit{O}), do not play a role

<table>
<thead>
<tr>
<th>v</th>
<th>s[v]</th>
<th>f[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>21</td>
<td>26</td>
</tr>
<tr>
<td>O</td>
<td>22</td>
<td>25</td>
</tr>
<tr>
<td>S</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

Finally we do \textsc{DFS-Visit(P)}. This one is quick. The adjacency list of \textit{P} consists only of \textit{S} which is already black. So
The sort is the list of vertices in the reverse order of their finish. In the algorithm when a vertex finishes we place it at the start of a linked list, initially nil. At the end, with negligible extra time, we have the list:

\[
\begin{array}{ccc}
v & s[v] & f[v] \\
P & 27 & 28 \\
\end{array}
\]

7. Let \( G \) be a DAG with a specific designated vertex \( v \). Uno and Dos play the following game. A token is placed on \( v \). The players alternate moves, Uno playing first. On each turn if the token is on \( w \) the player moves the token to some vertex \( u \) with \((w, u)\) an edge of the DAG. When a player has no move, he or she loses. Except for the first part below, we assume Uno and Dos play perfectly.

(a) Argue that the game must end.

Solution: Let \( G \) have \( V \) vertices. If the game went on for \( V \) moves the chip would hit \( V + 1 \) positions \( v = v_0, v_1, \ldots, v_V \) and so some position would be hit twice – some \( i < j \) with \( v_i = v_j \) – but that gives a cycle \( v_i v_{i+1} \cdots v_{j-1} v_i \).

(b) Define \( VALUE[z] \) to be the winner of the game (either Uno or Dos) where the token is initially placed at vertex \( z \) and Uno plays first. Suppose the \( VALUE[w] \) are known for all \( w \in Adj[z] \). How do those values determine \( VALUE[z] \).

Solution: Suppose there is some \( w \in Adj[z] \) with \( VALUE[w] \) equal Dos. Uno makes that move. Now, as the roles are reversed and Dos must move first so Uno wins. Therefore \( VALUE[z] \) is Uno. If there is no such \( w \) then whatever move Uno makes a position \( w \) is reached with \( VALUE[w] \) equal Uno. But this means the the player making the first move will win, and that player is Dos. Therefore \( VALUE[z] \) is Dos.

(c) Using the above idea modify DFS to find who wins the original game. Give an upper bound on the time of your algorithm.

Solution: Apply DFS-VISIT\([v]\) with an additional field \( VALUE \). We can implement the previous part in several ways. The easiest is to wait until a vertex \( z \) has become black. At that time check the \( VALUE \) (they will already have been determined) of all \( w \in Adj[z] \). If any is Dos, set \( VALUE[z] \) to be Uno, otherwise (this includes the case where \( Adj[z] \) is empty!) set \( VALUE[z] \) to be...
Dos. The time is $O(V + E)$. It could be considerably smaller than $V + E$ as $\text{DFS-\textsc{Visit}}[v]$ might only reach a small part of the graph.