## Review

- Recursive formula gives recursive algorithm - but algorithm often takes exponential time.
- 'Memoization' used to speed up algorithms.
- Store values already computed. At each call we check to see if value already computed.
- Optimal solution to problem gives optimal solution to subproblems.
- Dynamic programming used to optimize.
- DAGs and topological sorts.

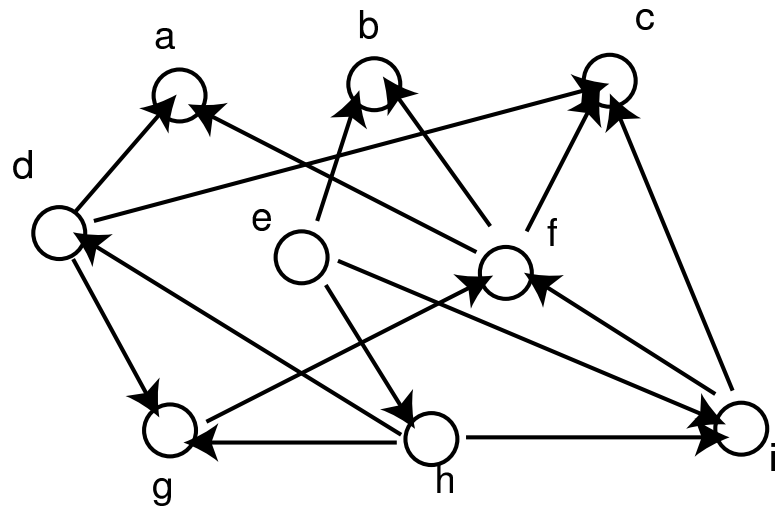
## Topological sort

A **topological sort** of a digraph is an ordering $v_1, v_2, \ldots, v_n$ of the vertex such that if $(v_i, v_j)$ is an edge then $i < j$.

We can use induction and the previous lemma to prove:

**Lemma**
*Every DAG has a topological sort*

Example 2.1:




Topological sort: $e, h, d, g, f, i, b, a, c$. There are others.

## Representing directed graphs

We use an **adjacency list** representation. For each vertex $u$ we list the vertices $v$ such that $(u, v) \in E$.

e.g. for Example 2.1:

| $v$ | $Adj[v]$ | $Pred[v]$ |
|---|---|---|
| $a$ | | |
| $b$ | | |
| $c$ | | |
| $d$ | $a, g$ | |
| $e$ | $b, h, i$ | |
| $f$ | $a, b, c$ | |
| $g$ | $f$ | |
| $h$ | $d, g, i$ | |
| $i$ | $c, f$ | |

From this, we can easily compute a **predecessor table**. For each $v$, find the list of vertices $u$ such that $(u, v) \in E$.

$FindPreds(G)$
1. Initialise $Prev[v] \leftarrow \emptyset$ for all $v \in V$.
2. **for** all $u \in V$ **do**
3. **for** all $v \in Adj[u]$ **do**
4. Add $u$ to the end of the list $Pred[v]$.

## Project scheduling

We wish to complete a project/computation in the shortest amount of time possible. We can perform tasks in parallel, but there are some tasks that we have to finish before we can begin others.

*Input*:

- Set of tasks $t_1, \ldots, t_n$.
- Set $P$ of constraints $(t_i, t_j)$. The pair $(t_i, t_j)$ means that task $t_i$ must be completed before task $t_j$ can begin.
- Task $t_i$ takes $f(t_i)$ time to run.

*Question*: If we are allowed unlimited parallel processors, how long does it take to complete all of the tasks?

## Example

| task | time | must be done after: |
|---|---|---|
| $A$ | 3 | $B$ |
| $B$ | 3 | |
| $C$ | 5 | |

We start with $B$ and $C$ and run $A$ after $B$ has finished. Total time is 6.

**Solution - recursion**

Let $g(t_i)$ denote the first possible time that $t_i$ could finish.

• If there are no precedent constraints on $t_i$ then we can run $t_i$ straight away, and $g(t_i) = f(t_i)$.

• If there are precedent constraints, then we have to wait until they have all finished before we can run $t_i$. Hence

$$g(t_i) = \max\{g(t_j) : (t_j, t_i) \in P\} + f(t_i)$$

1. Construct $G$ with vertices $\{t_1, \ldots, t_n\}$ and edge set $E = P$.
2. Construct the $Pred$ tables for $G$.
3. Initialise $g[t_i] \leftarrow -1$ for all $i$.
4. Call $FinishTime(t_i)$ for each $t_i$.
5. Return the maximum of $g[t_i]$ for $i = 1, 2, \ldots, n$.

$FinishTime(t_i)$

1. **If** $g[t_i] \geq 0$ **then**
2. **return** $g[t_i]$
3. **else**
4. **if** $Pred[t_i]$ is empty **then**
5. $g[t_i] \leftarrow f[t_i]$
6. **else**
7. $g[t_i] \leftarrow \max\left\{FinishTime(t_j) : t_j \in Pred[t_i]\right\} + f(t_i)$
8. **return** $g[t_i]$

## Longest path

How long is the longest path in a DAG? This is a hard problem in a regular digraph, but can be easily solved in a DAG.

Similar to project scheduling. Let $l(v)$ denote the longest path ending in vertex $v$. Then

- $l(v) = 0$ if $v$ is a source.
- Otherwise $l(v) = \max\{l(u) : u \in Pred[v]\} + 1$

1. Initialise $L[v] \leftarrow -1$ for all $v$.
2. Construct the $Pred$ tables for $G$.
4. Call $LongPath(v)$ for each $v$.
5. Return the maximum of $L[v]$ for $i = 1, 2, \ldots, n$.

$LongPath(v)$
1. **If** $L[v] \geq 0$ **then**
2. **return** $L[v]$
3. **else**
4. **if** $Pred[v]$ is empty **then**
5. $L[v] \leftarrow 0$
6. **else**
7. $L[v] \leftarrow 1 + \max\{LongPath(u) : u \in Pred[v]\}$
8. **return** $L[v]$.

## Features of Dynamic Programming

- Problem divided into *overlapping* sub-problems. (Compare divide and conquer, where subproblems are independent)
- The solution of a subproblem generally depends on solutions of further subproblems.
- A straight recursive solution leads to an exponential time algorithm.
- Solutions of subproblems are stored, as they are often used multiple times during computation.

We can denote dependencies between subproblems using a directed graph. Each subproblem corresponds to one vertex. An edge from subproblem $u$ to subproblem $v$ if we use the solution of $u$ in the solution of $v$.

Dynamic programming works if and only if this directed graph is acyclic.

## General problems in Dynamic Programming

1) How to avoid recursion? (function calls generally take a lot of time).

2) How to extract an optimal solution? (e.g. highest scoring path, longest path).

3) How to handle multiple optimals? (e.g. counting the number of optimal solutions).

## Avoiding recursion

Suppose that vertices in $G$ are labelled $v_1, v_2, \ldots, v_n$. A "simpler" alternative to $LongPath$ might be:

$WrongLongPath(G)$
1. **for** $i \leftarrow 1$ to $n$ **do**
2. **if** $v_i$ is a source **then**
3. $\quad L[v_i] \leftarrow 0$
4. **else**
5. $\quad L[v_i] \leftarrow 1 + \max\{L[v_j] : v_j \in Pred[v_i]\}$.

**Problem**: In step 5 we have no guarantee that the values $L[v_j]$ have already been initialised and computed.

**Solution**: First construct a topological sort $v_1, v_2, \ldots, v_n$ of the vertices. Then $v_j \in Pred[v_i]$ will imply that $(v_j, v_i) \in E$, so $v_j$ comes before $v_i$ in the ordering.

Principle:

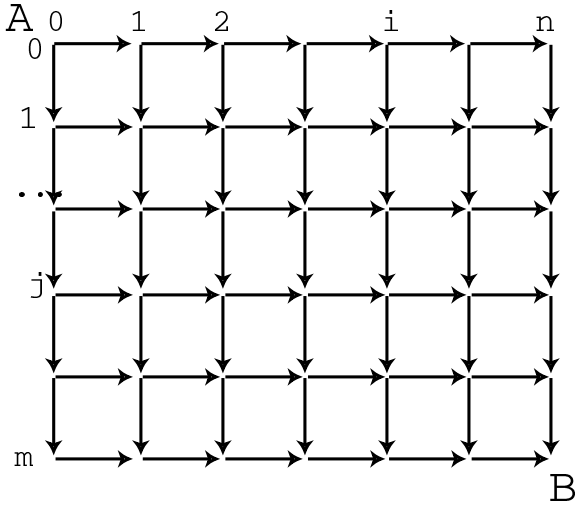
**If you are using loops with Dynamic Programming, make sure that subproblems are solved in a topological order, with respect to the dependency graph.**

## Example: flag collecting

Subproblems: one for each $(i, j)$. The subproblem is "What is the maximum score of a path from $(0, 0)$ to $(i, j)$ that always goes downwards or to the right"

To solve the problem for $(i, j)$ we need to have solved $(i - 1, j)$ and $(i, j - 1)$.

The dependency digraph is a directed version of the grid, with edges directed downwards and rightways.




One "topological sort" of this graph is given by the loops

**for** $i \leftarrow 0$ to $n$ **do**
  **for** $j \leftarrow 0$ to $m$ **do**
    Compute $G[i, j]$

**Recursion free algorithm for flag collecting**

$GoodPaths2$
1. **for** $i \leftarrow 0$ to $n$ **do**
2. **for** $j \leftarrow 0$ to $m$ **do**
3. **if** $i = 0$ and $j = 0$ **then**
4. $G[i,j] \leftarrow 0$
5. **else if** $i > 0$ and $j = 0$ **then**
6. $G[i,j] \leftarrow G[i-1,0] + H[i,0]$
7. **else if** $i = 0$ and $j > 0$ **then**
8. $G[i,j] \leftarrow G[0,j-1] + V[i,0]$
9. **else**
10. $G[i,j] \leftarrow \max\{G[i-1,j] + H[i,j], G[i,j-1] + V[i,j]\}$
11. **return** $G[n,m]$

Note: whenever we look-up $G[x,y]$ we know its already been computed.