Algorithm: $\text{DFS}(u)$:

- Initialize stack $S$ to be empty;
- Mark all vertices as unvisited; set $\text{Time} = 0$;
- Initialize search tree $T$ to be empty;
- Mark $u$ as visited and push $(S, u)$;
- $\text{pre}(u) = ++\text{Time}$;

while $S$ is not empty do

- $x = \text{peek}(S)$;
- \textbf{foreach} vertex $y \in \text{Adj}(x)$ \textbf{do}
  - \textbf{if} $y$ is not visited \textbf{then}
    - Mark $y$ as visited and push $(S, y)$;
    - $\text{pre}(y) = ++\text{Time}$;
  - \textbf{if} there is no such $y$ \textbf{then}
    - $\text{post}(x) = ++\text{Time}$; pop($S$);
A Recursive DFS

Algorithm: \texttt{DFS}(x):
\begin{itemize}
\item Mark \texttt{x} as visited;
\item set \texttt{pre}(x) = ++Time;
\item \textbf{foreach} \texttt{vertex} \texttt{y} \in \texttt{Adj}(x) \textbf{do}
  \begin{itemize}
  \item \textbf{if} \texttt{y} \textit{is not visited} \textbf{then}
    \begin{itemize}
    \item add edge \texttt{(x, y)} to \texttt{T};
    \item \texttt{DFS(y)};
    \end{itemize}
  \end{itemize}
\item set \texttt{post}(x) = ++Time;
\end{itemize}

Algorithm: \texttt{DFS}(G):
\begin{itemize}
\item Mark all vertices as unvisited;
\item Set \texttt{T} to be empty, \texttt{Time} = 0;
\item \textbf{while} \exists \textit{unvisited vertex} \texttt{u} \textbf{do}
  \begin{itemize}
  \item \texttt{DFS(u)};
  \end{itemize}
\item \textbf{return} \texttt{T}
\end{itemize}
Recursion

- Random Access Machines Model does not directly allow recursion
  - Neither does any real hardware

- Compilers will “roll out” recursive calls
  - Put all local variables of the calling procedure in a safe place
  - Execute the call
  - Return the result and restore the local variables

- Best data structure to implement a recursion: stack
  - LIFO does exactly the right thing
DFS Intuition

- exploring a maze
- from current vertex, move to another
- until you get stuck
- then backtrack till you find the first new possibility for exploration
DFS: An Example

- → tree edges
- → back edges
- → forward edges
- → cross edges
pre and post values

- Vertex $u$ is active in time interval $[\text{pre}(u), \text{post}(u)]$.

**Proposition**

For any two vertices $u$ and $v$, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are either disjoint, or one is contained in the other.

**Proof.**

1. Assume w.l.o.g that $\text{pre}(u) < \text{pre}(v)$, then vertex $u$ is visited before $v$.
2. If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
3. If $\text{DFS}(v)$ after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$. 
Edge Classification

Edges of $G$ can be classified into four categories:

1. **Tree edges**: edges that belong to $T$
2. **Back edges**: non-tree edges $(u, v)$ such that $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$.
3. **Forward edges**: non-tree edges $(u, v)$ such that $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$.
4. **Cross edges**: non-tree edges $(u, v)$ such that the intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint.

The values of $\text{pre}$ and $\text{post}$ are very useful in many applications.
A directed acyclic graph (DAG) is a directed graph without cycles.

- Generalize trees to directed graphs, with much richer structure
- Can be used to encode *precedence relations* or *dependence* in a natural way

A topological ordering of a DAG is an ordering of vertices as \(v_1, v_2, \ldots, v_n\), such that for every edge \((v_i, v_j)\), we have \(i < j\).

- All edges point “forward” in the ordering.

**Proposition**

G has a topological ordering if and only if G is a DAG.

**Question**

How to check if a given graph G is a DAG?
Lemma

Graph $G$ is a DAG if and only if there are no back edges in its DFS tree.

Proof.

$\Rightarrow$: If there is a back edge, then there is a cycle.

$\Leftarrow$: Assume there is a cycle $C$, let $u$ be the first vertex discovered on $C$.

- Let $(v, u)$ be the preceding edge in $C$. Then $v$ is a descendant of $u$.
- $(v, u)$ is a back edge.
Algorithm: Topological-Sort\((G)\):

Call DFS\((G)\) to compute post\((u)\) for each vertex \(u\);
return vertices in the reverse order of the post\((u)\);

Proof of Correctness

It suffices to show that for any \((u, v) \in E\), post\((v) < post(u)\).

- \((u, v)\) is a tree edge: \(v\) becomes a descendant of \(u\), post\((v) < post(u)\).
- \((u, v)\) is a forward/cross edge: post\((v) < post(u)\).

Proposition

Topological-Sort\((G)\) returns a topological ordering of a DAG in linear time.
DFS Summary

- Another linear time graph traversal algorithm.
- Helps to understand the structure of directed graphs.
- Can be used in topological sorting and many other applications.
Greedy Algorithms
What is a Greedy Algorithm

“Greed ... is good. Greed is right. Greed works.”


Hard (if not impossible) to have a precise definition.

An algorithm is **greedy** if it:

- builds up a solution in small steps without backtracking
- chooses a decision at each step to improve some local or current state
Pros and Cons

Pros:
- Easy to implement and often run fast.
- Imply interesting and useful structure of the problem.
- Lead to a first-cut heuristic when problems are not well understood.

Cons:
- **Very often** greedy algorithms don’t work.
- Easy to invent greedy algorithms for almost any problem; finding the right one and prove its correctness is the challenging task.

Every greedy algorithm needs a proof of correctness.
Interval Scheduling

**Input:** a set of jobs, job $j$ start at $s_j$ and finishes at $f_j$

Two jobs are **compatible** if they don’t overlap.

**Goal:** find a maximum subset of mutually compatible jobs.
Algorithm: `SomeGreedyAlgorithm(G)`:  
Let $R$ be the set of all jobs.;  
Initialize $X = \emptyset$;  
// $X$ will store all scheduled jobs  
while $R$ is not empty do  
    choose $j \in R$;  
    add $j$ to $X$;  
    remove from $R$ all jobs that overlap with $j$;  
return $X$

Main task: Decide the order in which to process jobs in $R$
Possible orders:

**Earliest start time:** Consider jobs in ascending order of $s_j$.

**Earliest finish time:** Consider jobs in ascending order of $f_j$.

**Shortest interval time:** Consider jobs in ascending order of $f_j - s_j$.

**Fewest conflicts:** For each job $j$, count the number of conflicting jobs $c_j$. Schedule in ascending order of $c_j$. 
Counter Examples

Figure: counterexample for earliest start time

Figure: counterexample for shortest interval

Figure: counterexample for fewest conflicts
Optimal Greedy Algorithm

**Algorithm:** SomeGreedyAlgorithm($G$):

Let $R$ be the set of all jobs.;
Initialize $X = \emptyset$ ; // $X$ will store all scheduled jobs

**while** $R$ is not empty **do**

choose $j \in R$ with the smallest $f_j$;
add $j$ to $X$;
remove from $R$ all jobs that overlap with $j$;

**return** $X$

**Theorem**

The greedy algorithm that picks jobs in the ascending order of their finishing times is optimal.