# ECE 374 B: Algorithms and Models of Computation, Spring 2022
# Midterm 2 – November 01, 2022

---

- **You will have 75 minutes (1.25 hours) to solve 4 problems. Most have multiple parts.** Don't spend too much time on questions you don't understand and focus on answering as much as you can!
- *No* resources are allowed for use during the exam except a multi-page cheatsheet and scratch paper on the back of the exam. ***Do not tear out the cheatsheet or the scratch paper!*** It messes with the auto-scanner.
- You should write your answers *completely* in the space given for the question. We will not grade parts of any answer written outside of the designated space.
- Please bring (sharpened) ***pencils and an eraser*** to take your exam with, unless you are *absolutely sure* you will not need to erase. We will *not* provide any additional scratch paper if you write in pen and make mistakes, nor will we provide pencils and erasers.
- Incorrect algorithms will receive a score of 0, but slower than necessary but correct algorithms will *always* receive some points, even brute force ones. Thus, *you should prioritize the correctness of your submitted algorithms over speed*; you will receive more points that way. On the other hand, submit the fastest algorithms that you know are correct; faster algorithms will receive more points.
- Any recursive backtracking algorithm or dynamic programming algorithm given without an *English* description of the recursive function (i.e., a description of the output of the function *in terms of their inputs*) will receive a score of 0.
- Any greedy algorithm or a modification of a standard graph algorithm given without a proof of correctness will receive a score of 0.
- Any algorithms written in actual code instead of pseudocode will receive a score of 0.
- For problems with a graph given as input, you may assume the graph is simple (i.e., it has no self-loops or parallel edges).
- Unless explicitly mentioned, **a runtime analysis is required for each given algorithm**.
- ***Don't cheat.*** If we catch you, you will get an F in the course.
- ***Good luck!***

---

Name: ______________________________

NetID: ______________________________

Date: ______________________________

## 1 Short answer (4 questions) - 26 points

For each question, there is only ***one*** correct answer. Circle the letter corresponding to the correct answer; any question with two or more choices marked will be graded as incorrect.

(a) What is the value of the following summation?

$$\sum_{k=0}^{n-1} 2^k = ?$$

**Solution:** Consider this problem in binary. Let $x^{(m)}$ be $x$ repeated $m$ times. In binary $2^m$ is $10^{(m-1)}$. This means, in binary, $\sum_{k=0}^{n-1} = 1^{(n-1)}$. $1^{(n-1)} + 1 = 10^{(n-1)}$, therefore $\sum_{k=0}^{n-1} = 2^n - 1$. ■

**Solution:** Alternative induction proof. First $\sum_{k=0}^{1-1} 2^k = 2^0 = 1 = 2^1 - 1$. Then supposed for the purpose of induction $\sum_{k=0}^{p-1} 2^k = 2^p - 1$ for all $p < n-1$. Then $\sum_{k=0}^{n-1} 2^k = 2^{n-1} + \sum_{k=0}^{n-2} 2^k = 2^{n-1} + 2^{n-1} - 1 = 2(2^{n-1}) - 1 = 2^n - 1$. Therefore, by induction, $\sum_{k=0}^{n-1} 2^k = 2^n - 1$. ■

**For more information: :** You likely remember this summation from Lecture 10 and HW6P3b. ■

(b) I formulated the solution to a particular question using the following recurrence:

$$f(x, y) = f(x, y-1) + f(x-1, y-1)$$
$$\textbf{Base:} \qquad f(x, 1) = 1 \qquad f(1, y) = 1$$

Using memoization, what is the optimal runtime of this algorithm?

**Solution:** To find $f(x, y)$ We want to compute an array, $A$ size $x$ by $y$, where $A[n, m] = f(n, m)$. Because $f(x, y) = f(x, y-1) + f(x-1, y-1)$ we compute the array in the order $A[1, 1], A[1, 2], \ldots, A[1, y], A[2, 1], \ldots, A[2, y], \ldots, A[x, y]$ so that all values necessary are already computed and can be pulled from the array. When filling out this array, the entry is either a base case in which case we input 1, or it is the sum of 2 previous cases which we pull from the array and add together. Both of these cases take constant time, $O(1)$, so the runtime is $xyO(1) = O(xy)$. ■

**For more information: :** For a more detail we solved a similar problem as a pre-lecture-teaser in Lecture 14. ■

(c) I'd like to use the median-of-medians algorithm but I don't want to write a function that finds the median value in a list of 5 values. Instead I break the input area into lists of 3 values, and choose the median of medians pivot that way. Evaluate the running time of this resulting algorithm. Does the running time increase or decrease?

**Solution:** Let $b$ be the MoM where we use length 3 lists. Half of the lists have 2 values less than $b$, except for the list where $b$ is the median, which only has 1 value less than $b$. This means the number of elements less than $b$ is $\geq n/3$. This means $|A_{\text{greater}}|$ and $|A_{\text{less}}| \leq 2n/3$. This means $T(n) \leq T(n/3) + T(2n/3) + O(n)$. $n/3 + 2n/3 = n$ so the work done at each level of the recursion tree is the same at $O(n)$. There are $log(n)$ levels of the recursion tree so the runtime is $O(nlog(n))$. This is a longer runtime than length 5 lists which has runtime $O(n)$, so the runtime increases. ■

**For more information: :** We went over this problem exhaustively in Lecture 12 (almost half the lecture was devoted to this). ■

(d) For any number of vertices $n$, describe a graph with $n$ vertices that has the maximum number of topological sorts. Recall that a topological sort for a directed acyclic graph $G$ is an ordering of the vertices of $G$ such that for every edge $a \to b \in G$, $a$ comes before $b$ in the sequence.

**Solution:** Let $G = (V, E)$ where $|V| = n$ and $E = \varnothing$. There are $n!$ unique topological sorts of $G$, which is the maximum number of distinct sorts of $n$ elements. ■

**For more information: :** We discussed this very problem in Lecture 16. ■

## 2 Recursion - 20 points

I need to use the Depth-First-Search (DFS) algorithm but I have no standard library files that implement stacks/queues. Instead of writing my own stack structure, I decide it'll be easier to use the system stack to implement DFS. In other words, I'll implement a recursive algorithm. Write the recursive version of Depth-First-Search.

**Solution:** The idea with Depth-First-Search is to go down one "branch" within the tree and then after visiting all nodes in that branch go to to next branch. Because we want to do this for each node we visit, this can be written as recursion problem. Another way of thinking about this traversal problem is that we want to go to the deepest node, and once we have done that, we check for its silbings, and if there are none we backtrack to the parent node.

```
DFS Recursive($v$):
  if v is unmarked
      mark $v$
      for each edge$v \longrightarrow w$
          DFS Recursive($w$)
```

This can be made slightly faster by checking if a vertex has already been visited:

```
DFS Recursive($v$):
  mark $v$
  for each edge$vw$
      if $w$ is unmarked
          parent($w$) $\leftarrow v$
          DFS Recursive($w$)
```

This algorithm runs in $O(V+E)$ time, where $V$ is the number of vertices, and $E$ is the number of edges. ■

**For more information:** : We go over this in one of the pre-lecture-teasers (Lecture 16). The actual pseudo-code is peppered throughout that lecture. ■

## 3 DP problem - 28 points

1. **Largest Subsequence Product [12 Points]** You are given as input an array of integers $A[1..n]$. For a given subsequence $A'$ of size $m$, the *subsequence product* is $\prod_{j \in 1..m} A'[j]$. Give a dynamic programming that finds the largest subsequence product.

   For example, given $A = [-1, -2, -1, 3]$, the largest product of a subsequence is 6.

   *Hint: linear time is possible.*

**Solution:** This problem is related to the Longest Increasing Subsequence problem discussed in class and homework 6 problem 2. By convention, the product of an empty subsequence is 1.

The challenge is that we need to consider negative numbers. The largest subsequence could be the product of the *smallest subsequence* and a negative number.

Consider $i \in [1, ..., n]$, let $LSP(i, 0)$ be the largest subsequence product in $A[1..i]$, let $LSP(i, 1)$ be the smallest subsequence product in $A[1..i]$. Then $LSP(i, 0)$ and $LSP(i, 1)$ satisfy the following recurrence:

$$LSP(i,0) = \begin{cases} 1 & \text{if } i = 0 \\ \max \begin{Bmatrix} LSP(i-1,0) \\ LSP(i-1,0) \cdot A[i] \\ LSP(i-1,1) \cdot A[i] \end{Bmatrix} & \text{otherwise} \end{cases}$$

$$LSP(i,1) = \begin{cases} 1 & \text{if } i = 0 \\ \min \begin{Bmatrix} LSP(i-1,1) \\ LSP(i-1,0) \cdot A[i] \\ LSP(i-1,1) \cdot A[i] \end{Bmatrix} & \text{otherwise} \end{cases}$$

Thus, the largest subsequence product of the array A is $LSP(n, 0)$.

We can memoize the function $LSP$ into an array $LSP[0..n][0..1]$.

```
LSP($A[1..n]$):
  initialize LSP[$0..n$][$0, 1$]
  for $i \leftarrow 0$ to $n$:
      if $i = 0$:
          LSP[$i$][0] $\leftarrow$ 1
          LSP[$i$][1] $\leftarrow$ 1
      else:
          LSP[$i$][0] $\leftarrow \max\{LSP[i-1][0], LSP[i-1][0] \cdot A[i], LSP[i-1][1] \cdot A[i]\}$
          LSP[$i$][1] $\leftarrow \min\{LSP[i-1][1], LSP[i-1][0] \cdot A[i], LSP[i-1][1] \cdot A[i]\}$
  return LSP[$n$][0]
```

This algorithm has $O(n)$ time complexity and $O(n)$ space complexity. $O(1)$ space complexity can be achieved by optimizing the memoization structure. ■

**For more information: :** Very similar to HW6-P2. ■

2. **Outputting the subsequence. [4 points]** Describe how to modify your algorithm to output a subsequence that has the largest product. For example, given $A$ above, you could output $A' = [2, -1, 3]$. Analyze the runtime of the resulting algorithm.

   *Hint: Your answer must describe what additional information must stored in the memo table. It must also give iterative or recursive pseudocode to output the result from the table.*

**Solution:** To output a subsequence that has the largest product, we need to add two True/False tables. Consider $i \in [1, ..., n]$, let $LSP(i, 2)$ be if A[$i$] contributes to the largest subsequence product, let $LSP(i, 3)$ be if A[$i$] contributes to the smallest subsequence product. The algorithm goes as follows:

```
LSP(A[1..n]):
    initialize LSP[0..n][0..3]
    for $i \leftarrow 0$ to $n$:
        if $i = 0$:
            LSP[$i$][0] $\leftarrow 1$
            LSP[$i$][1] $\leftarrow 1$
        else:
            LSP[$i$][0] $\leftarrow$ max{LSP[$i-1$][0], LSP[$i-1$][0]·A[$i$], LSP[$i-1$][1]·$A[i]$}
            LSP[$i$][1] $\leftarrow$ min{LSP[$i-1$][1], LSP[$i-1$][0]·$A[i]$, LSP[$i-1$][1]·$A[i]$}
    for $i \leftarrow 1$ to $n$:
        if LSP[$i$][0] = LSP[$i-1$][0]:
            LSP[$i$][2] $\leftarrow$ False
        if LSP[$i$][0] = LSP[$i-1$][0]·$A[i]$:
            LSP[$i$][2] $\leftarrow$ True
        if LSP[$i$][0] = LSP[$i-1$][1]·$A[i]$:
            for $j \leftarrow 1$ to $i$:
                LSP[$j$][2] $\leftarrow$ LSP[$j$][3]
            LSP[$i$][2] $\leftarrow$ True
        if LSP[$i$][1] = LSP[$i-1$][1]:
            LSP[$i$][3] $\leftarrow$ False
        if LSP[$i$][1] = LSP[$i-1$][1]·$A[i]$:
            LSP[$i$][3] $\leftarrow$ True
        if LSP[$i$][1] = LSP[$i-1$][0]·$A[i]$:
            for $j \leftarrow 1$ to $i$:
                LSP[$j$][3] $\leftarrow$ LSP[$j$][2]
            LSP[$i$][3] $\leftarrow$ True
    for $i \leftarrow 1$ to $n$:
        if LSP[$i$][2] = True:
            Output A[$i$]
```

In the worst case, when all the integers in A[$i$] are smaller than $-1$, LSP$(i, 0) \leftarrow$ LSP$(i-1, 1)\cdot$ A[$i$] and LSP$(i, 1) \leftarrow$ LSP$(i-1, 0)\cdot$ A[$i$] for all $i$'s. In this case, the algorithm takes O($n^2$) time to run. ■

**For more information: :** We go over how to get the actual sequence using predecessor variables in nearly all the the graph algorithms discussed. Lecture 18 is very explicit about it. ■

3. **[Largest Subsequence Dot Product [12 points]** You are given a pair of same size input arrays of integers, $A[1..n], B[1..n]$. Given a pair of subsequences $A', B'$ of the same size $m$, the *subsequence dot product* is $\sum_{j \in 1..m} A'[j] * B'[j]$.

   For example, given $A = [-1, 2, -1], B = [2, -1, 1]$, the largest dot product is 5.

   Give a dynamic programming solution to find the largest subsequence dot product.

   *Hint: quadratic time is possible.*

**Solution:** Let $LSDP(i, j)$ denote the largest subsequence dot product of $A[i..n]$ and $B[j..n]$. then, $LSDP$ follows the recurrence below.

$$LSDP(i,j) \begin{cases} 0 & \text{if } i > n \text{ or } j > n \\ \max \begin{cases} LSDP(i+1, j) \\ LSDP(i, j+1) \\ A[i]B[j] + LSDP(i+1, j+1) \end{cases} & \text{otherwise} \end{cases}$$

We can memoize the function $LSDP$ into an array $LSDP[1..n+1, 1..n+1]$. We can fill the array in a nested loop of $i, j$, both from $n$ to 1. Finally, $LSDP[1, 1]$ has the largest subsequence dot product.

```
LSDP($A[1..n]$):
  for $i \leftarrow 1$ to $n+1$:
      LSDP[$i, n+1$] $\leftarrow 0$
      LSDP[$n+1, i$] $\leftarrow 0$
  for $j \leftarrow n$ down to 1:
      for $i \leftarrow n$ down to 1:
          LSDP[$i, j$] $\leftarrow \max\{$LSDP[$i+1, j$], LSDP[$i, j+1$], $A[i]B[j]$ + LSDP[$i+1, j+1$]$\}$
  return LSDP[1, 1]
```

The running time of the algorithm is $O(n^2)$. ■

**For more information: :** This is extremely similar to the Longest common subsequence discussed in Lecture 14. ■

## 4 Graph algorithms - 26 points

(a) Derive an efficient algorithm to find all the source vertices in a directed graph. Recall that a source vertex has no incoming edges.

**Solution:** We need to count the number of incoming edges from each vertex in the graph. The numbers of incoming edges for all the vertices are initialized as 0 at first. Next, we loop through all the vertices. For each vertex $u$, we can get the list of outgoing edges using $\text{Out}(u)$. For each outgoing edge $(u, v)$, the number of incoming edges for vertex $v$ would increase by 1.

Finally, we can check the number of incoming edges for each vertex. For the vertices with no incoming edges, they are source vertices. This takes $O(V + E)$ time since we go through all the vertices and edges once. ■

**For more information:** : This is very similar to our first topological sort formulation (Lecture 16). ■

(b) Give an efficient algorithm that determines if a particular weighted directed graph has a negative cycle.

**Solution:** Add a new vertex $s'$ and connect it to all nodes of $G$ with zero length edges. The new graph is $G'$. To check for a negative cycle reachable from vertex $s'$, we run Bellman-Ford algorithm on $G'$ starting at $s'$. After the main loop of the Bellman-Ford algorithm, the shortest distance should be guaranteed without negative cycles. Therefore, after the main loop of the Bellman-Ford algorithm, we check once again if any edge in $G'$ is tense. The edge $(u, v)$ is tense if $d(u) + l(u, v) < d(v)$. If there is a tense edge, report that there is a negative cycle in $G$. Otherwise, report that there is not.

This takes $O(E)$ time because checking if a given edge is tense takes $O(1)$ time, so the runtime of the entire algorithm is dominated by the runtime of Bellman-Ford, which is $O(VE)$. ■

**Solution:** We can run Floyd-Warshall on $G$. The distance from $v$ to $v$ should be zero without negative cycles. After that, for each vertex $v$, we can check if the distance from $v$ to $v$ found by Floyd-Warshall is negative. If so, report that there is a negative cycle in $G$. Otherwise, report that there is not.

The check takes $O(V)$ time for all vertices, so the runtime of the algorithm is dominated by the runtime of Floyd-Warshall, which is $O(V^3)$. ■

**For more information:** : If you chose to do BF, we discussed graph manipulation in the first part of Lecture 18 (pre-teaser). Floyd-Warshall is not required but is available in the materials and was listed in the skillset. So multiple ways to get a correct anser depending on your study style (lectures vs books). ■

*This page is for additional scratch work!*

# ECE 374 B Algorithms: Cheatsheet

## 1 Recursion

### Simple recursion

**Definitions**

- **Reduction:** solve one problem using the solution to another.
- **Recursion:** a special case of reduction - reduce problem to a *smaller* instance of *itself* (self-reduction).
  - Problem instance of size $n$ is reduced to *one or more* instances of size $n-1$ or less.
  - For termination, problem instances of small size are solved by some other method as *base cases*

**Tower of Hanoi**

Arguably the most famous example of recursion. The goal is to move $n$ disks one at a time from the first peg to the last peg.

Pseudocode: Tower of Hanoi

```
Hanoi($n$, src, dest, tmp):
  if ($n > 0$) then
    Hanoi($n-1$, src, tmp, dest)
    Move disk $n$ from src to dest
    Hanoi($n-1$, tmp, dest, src)
```

### Recurrences

Suppose you have a recurrence of the form $T(n) = rT(n/c) + f(n)$.

The *master theorem* gives a good asymptotic estimate of the recurrence. If the work at each level is:

| | | |
|---|---|---|
| Decreasing: | $rf(n/c) = \kappa f(n)$ where $\kappa < 1$ | $T(n) = O(f(n))$ |
| Equal: | $rf(n/c) = f(n)$ | $T(n) = O(f(n) \cdot \log_c n)$ |
| Increasing: | $rf(n/c) = Kf(n)$ where $K > 1$ | $T(n) = O(n^{\log_c r})$ |

Some useful identities:

- Sum of integers: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- Geometric series closed-form formula: $\sum_{k=0}^{n} ar^k = \frac{1-r^{n+1}}{1-r}$
- Logarithmic identities: $\log(ab) = \log a + \log b$, $\log(a/b) = \log a - \log b$, $a^{\log_c b} = b^{\log_c a}$ $(a, b, c > 1)$.

### Backtracking

*Backtracking* is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn't lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

Pseudocode: LIS - Naive enumeration

```
algLISNaive($A[1..n]$):
  maxmax $= 0$
  for each subsequence $B$ of $A$ do
    if $B$ is increasing and $|B| >$ max then
      $max = |B|$
  return max
```

On the other hand, we don't need to generate every subsequence; we only need to generate the subsequences that are increasing:

Pseudocode: LIS - Backtracking

```
LIS_smaller($A[1..n], x$):
  if $n = 0$ then return $0$
  max $= LIS\_smaller(A[1..n-1], x)$
  if $A[n] < x$ then
    max $= \max\{$max$, 1 + LIS\_smaller(A[1..(n-1)], A[n])\}$
  return max
```

### Divide and conquer

*Divide and conquer* is an algorithm paradigm involving the decomposition of a problem into the same subproblem, solving them separately and combining their results to get a solution for the original problem.

**Sorting algorithms**

| Algorithm | Runtime | Space |
|---|---|---|
| Mergesort | $O(n \log n)$ | $O(n \log n)$<br>$O(n)$ (if optimized) |
| Quicksort | $O(n^2)$<br>$O(n \log n)$ if using MoM | $O(n)$ |

**Karatsuba's algorithm**

We can divide and conquer multiplication like so:

$$bc = 10^n b_L c_L + 10^{n/2}(b_L c_R + b_R c_L) + b_R c_R.$$

We can rewrite the equation as:

$$\begin{aligned} bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) &= (b_L c_L)x^2 \\ &+ \Big((b_L + b_R)(c_L + c_R) - b_L c_L - b_R c_R\Big)x \\ &+ b_R c_R, \end{aligned}$$

Its running time is $O(n^{\log_2 3}) = O(n^{1.585})$.

### Linear time selection

The *median of medians* (MoM) algorithms give a element that is larger than $\frac{3}{10}$'s and smaller than $\frac{7}{10}$'s of the array elements. This is used in the linear time selection algorithm to find element of rank $k$.

Pseudocode: Quickselect with median of medians

```
Median-of-medians($A$, $i$):
  sublists = [A[j:j+5] for j $\leftarrow 0, 5, \ldots,$ len($A$)]
  medians = [sorted(sublist)[len(sublist)/2]
            for sublist $\in$ sublists]

  // Base case
  if len(A) $\leq$ 5 return sorted(a)[i]

  // Find median of medians
  if len(medians) $\leq$ 5
    pivot = sorted(medians)[len(medians)/2]
  else
    pivot = Median-of-medians(medians, len/2)

  // Partitioning step
  low = [j for j $\in A$ if $j < pivot$]
  high = [j for j $\in A$ if $j > pivot$]

  k = len(low)
  if i < k
    return Median-of-medians(low, i)
  else if i > k
    return Median-of-medians(low, i-k-1)
  else
  return pivot
```

## Dynamic programming

*Dynamic programming* (DP) is the algorithm paradigm involving the computation of a recursive backtracking algorithm iteratively to avoid the recomputation of any particular subproblem.

### Longest increasing subsequence

The longest increasing subsequence problem asks for the length of a longest increasing subsequence in a unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

$$\mathit{LIS}(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ \mathit{LIS}(i-1,j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} \mathit{LIS}(i-1,j) \\ 1 + \mathit{LIS}(i-1,i) \end{cases} & \text{else} \end{cases}$$

Pseudocode: LIS - DP

```
LIS-Iterative($A[1..n]$):
  $A[n+1] = \infty$
  for $j \leftarrow 0$ to $n$
    if A[i] $\leq$ A[j] then $LIS[0][j] = 1$

  for $i \leftarrow 1$ to $n-1$ do
    for $j \leftarrow i$ to $n-1$ do
      if $A[i] \geq A[j]$
        $LIS[i,j] = LIS[i-1,j]$
      else
        $LIS[i,j] = \max\{LIS[i-1,j],$
          $1 + LIS[i-1,i]\}$
  return $LIS[n, n+1]$
```

### Edit distance

The edit distance problem asks how many edits we need to make to a sequence for it to become another one. The recurrence is given as:

$$\mathrm{Opt}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1,j-1), \\ \delta + \mathrm{Opt}(i-1,j), \\ \delta + \mathrm{Opt}(i,j-1) \end{cases}$$

**Base cases:** $\mathrm{Opt}(i,0) = \delta \cdot i$ and $\mathrm{Opt}(0,j) = \delta \cdot j$

Pseudocode: Edit distance - DP

```
$EDIST(A[1..m], B[1..n])$
  for $i \leftarrow 1$ to $m$ do $M[i,0] = i\delta$
  for $j \leftarrow 1$ to $n$ do $M[0,j] = j\delta$

  for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
      $M[i][j] = \min \begin{cases} COST[A[i]][B[j]] \\ \quad + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$
```

# 2 Graph algorithms

## Graph basics

A graph is defined by a tuple $G = (V, E)$ and we typically define $n = |V|$ and $m = |E|$. We define $(u, v)$ as the edge from $u$ to $v$. Graphs can be represented as **adjacency lists**, or **adjacency matrices** though the former is more commonly used.

- *path*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path). *Note:* a single vertex $u$ is a path of length 0.
- *cycle*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. A single vertex is not a cycle according to this definition.
  *Caveat:* Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term *tour*.
- A vertex $u$ is *connected* to $v$ if there is a path from $u$ to $v$.
- The *connected component* of $u$, $\mathrm{con}(u)$, is the set of all vertices connected to $u$.
- A vertex $u$ can *reach* $v$ if there is a path from $u$ to $v$. Alternatively $v$ can be reached from $u$. Let $rch(u)$ be the set of all vertices reachable from $u$.

## Directed acyclic graphs

Directed acyclic graphs (dags) have an intrinsic ordering of the vertices that enables dynamic programming algorithms to be used on them.
A *topological ordering* of a dag $G = (V, E)$ is an ordering $\prec$ on $V$ such that if $(u, v) \in E$ then $u \prec v$.

Pseudocode: Kahn's algoritm

```
TP-sort-list = []
Count in-degree for each vertex
while $v \in V$ that has in$(v) = 0$:
  Add v to TP-sort-list
  Remove $v$ from $V$
  for $v$ in $(u, v) \in E$:
    in$(v) \leftarrow$ in$(v) - 1$
return TP-sort-list
```

**Running time:** $O(n + m)$

- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-visit order.

## DFS and BFS

Pseudocode: Explore (DFS/BFS)

```
Explore($G$,$u$):
  array Visited$[1..n]$
  Initialize: Visited$[i] \leftarrow$ False for $i = 1, \ldots, n$
  Bag data structure: B
  Add $u$ to ToExplore and to $S$, Visited$[u] \leftarrow$ True
  Make tree $T$ with root as $u$
  while B is non-empty do
    Remove node $x$ from B
    for each edge $(x, y)$ in $Adj(x)$ do
      if Visited$[y]$ = False
        Visited$[y] \leftarrow$ True
        Add $y$ to B
        Add $y$ to $S$
        Add $y$ to $T$ with $x$ as its parent
```

Note:

- If B is a queue, *Explore* becomes BFS.
- If B is a stack, *Explore* becomes DFS.

---

**Pre/post numbering**

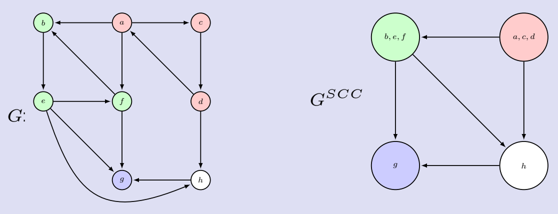
Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if a edge $(u, v)$ is a:

- *Forward edge*: pre$(u)$ < pre$(v)$ < post$(v)$ < post$(u)$
- *Backward edge*: pre$(v)$ < pre$(u)$ < post$(u)$ < post$(v)$
- *Cross edge*: pre$(u)$ < post$(u)$ < pre$(v)$ < post$(v)$

## Strongly connected components

- Given a directed graph $G$, $u$ is *strongly connected* to $v$ if $v \in$ rch$(u)$ *and* $u \in$ rch$(v)$.
- A group of vertices that are all strongly connected to one-another is called a strongly connected component.

**Example of Meta-graph:**



Pseudocode: Meta-Graph - Linear Time

```
do $(G^{rev})$ and output vertices in decreasing post order.
Mark all nodes as unvisited
for each $u$ in the computed order do
  if $u$ is not visited then
    $(u)$
    Let $S_u$ be the nodes reached by $u$
    Output $S_u$ as a strong connected component
    Remove $S_u$ from $G$
```

## Shortest paths

**Dijkstra's algorithm:**
Find minimum distance from vertex $s$ to **all** other vertices in graphs *without* negative weight edges.

Pseudocode: Dijkstra

```
Initialize for each node v, $d(s, v) = \infty$
Initialize $X = \varnothing$, $d(s, s) = 0$
for $i = 1$ to $n$ do
  Let $v$ be such that $d(s, v) = \min_{u \in V-X} d(s, u)$
  $X = X \cup \{v\}$
  for each $u$ in Adj$(v)$ do
    $d(s, u) = \min\{d(s, u),\ d(s, v) + \ell(v, u)\}$
```

**Running time:** $O(m + n\log n)$ (if using a Fibonacci heap as the priority queue)

---

**Bellman-Ford algorithm:**
Find minimum distance from vertex $s$ to **all** other vertices in graphs *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(v, k) = \min \begin{cases} \min_{u \in V}(d(u, k-1) + \ell(u, v)). \\ d(v, k-1) \end{cases}$$

**Base cases:** $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$.

Pseudocode: Bellman-Ford

```
for each $u \in V$ do
  $d(u) \leftarrow \infty$
$d(s) \leftarrow 0$

for $k = 1$ to $n - 1$ do
  for each $v \in n$ do
    for each edge $(u, v) \in in(v)$ do
      $d(v) = \min\{d(v), d(u) + c(u, v)\}$

for each $v \in V$ do
    $Dist(s, v) \leftarrow d(v)$
```

**Running time:** $O(nm)$

---

**Floyd-Warshall algorithm:**
Find minimum distance between any two vertices $i$ and $j$. It is a DP algorithm with the following recurrence:

$$d(i, j, k) = \min \begin{cases} d(i, j, k-1) \\ d(i, k, k-1) + d(k, j, k-1) \end{cases}$$

**Base cases:** $dist(i, j, 0) = \ell(i, j)$ if $(i, j) \in E$, otherwise $\infty$

Pseudocode: Floyd-Warshall

```
for $i \leftarrow 1$ to $n$ do
  for $j \leftarrow 1$ to $n$ do
    $d(i, j, 0) \leftarrow \ell(i, j)$
 (* $\ell(i, j) \leftarrow \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

for $k \leftarrow 1$ to $n$ do
  for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
      $d(i, j, k) \leftarrow \min \begin{cases} d(i, j, k-1), \\ d(i, k, k-1) + d(k, j, k-1) \end{cases}$
for $i \leftarrow 1$ to $n$ do
  if $d(i, i, n) < 0$ then
    Output "$\exists$ negative cycle in $G$"
```

**Running time**: $\Theta(n^3)$