CS/ECE-374-B: Algorithms and Models of Computation, Spring 2024

Midterm exam 2 – March 26, 2024

- You can do hard things! Grades do matter, but not as much as you may think, but then life is uncertain anyway, so what?
- **Don't cheat.** The consequence of cheating is far greater than the reward. Just try your best and you'll be fine.
- **Please read the entire exam before writing anything.** There are 6 problems and most have multiple parts.
- This is a closed-book exam. At the end of the exam, you'll find a multi-page cheat sheet. Do not tear out the cheat sheet! No outside material is allowed on this exam.
- You should write your answers legibly and in the space given for the question. Overly verbose answers will be penalized.
- Scratch paper is available on the back of the exam. Do not tear out the scratch paper! It messes with the auto-scanner.
- **You have** 75 **minutes (**1.25 **hours) for the exam.** Manage your time well. *Do not spend too much time on questions you do not understand and focus on answering as much as you can!*
- We know that this exam is **shorter in length** compared to the first one. Make sure you *use the time well to think, be precise, and show as much work as possible*.

Name: _____

NetID: _____

Date: _____

Problem 1 [10 points]

For each of the following statements, answer if it is `True` or `False`. Use the table at the bottom to mark your choices.

i. Recursion is a special case of reduction that includes reducing the problem into smaller instances of itself.

ii. Merging two sorted arrays each of size n into a single sorted array requires a minimum $O(n \log n)$ time.

iii. If topological sort exists for a graph then that graph has a cycle.

iv. If for some two nodes, the pre-post numbering intervals in DFS are disjoint then it means that the graph is disconnected.

v. Checking if a sequence is increasing or not can be achieved in a minimum of $O(n)$ time.

vi. The node with maximum post numbering in DFS is in a sink strongly connected component of the original graph.

vii. Every directed acyclic graph has either a source or a sink but not both.

viii. The asymptotic runtime of Merge Sort depends on the number of splits one makes of the original array.

ix. If a graph has a cycle then there is a back-edge in its DFS.

x. Decreasingly sorted pre-numberings in DFS give a topological sort of the given directed acyclic graph.

Table 1.

Statement	Your choice
i.	`True`
ii.	`False`
iii.	`False`
iv.	`False`
v.	`True`
vi.	`False`
vii.	`False`
viii.	`False`
ix.	`True`
x.	`False`

Problem 2 [15 points]

Solve the following recurrence relations *exactly*, i.e, obtain a closed form formula for $f(n)$ without order terms/bounds.

Useful formula: $\sum_{k=0}^{n} ar^k = \frac{a(1-r^{n+1})}{1-r}$.

a. $f(n) = f(n-1) + n2^n,\ n > 0$ and $f(0) = 3$.

Solution: You can iteratively replace $f(\cdot)$ on the right using the given recursive equation until you get to the base case. At that point use, the geometric series formula. Note that you need to take care of a telescopic sum before using the geometric series formula.
$f(n) = f(n-1) + n2^n$
$f(n) = f(n-2) + (n-1)2^{n-1} + n2^n$
$f(n) = f(0) + 1.2 + 2.2^2 + ... + n2^n$
$f(n) = f(0) + S(n)$

$S(n) = \sum_{k=1}^{n} k2^k = 1.2 + 2.2^2 + ... + n2^n$ –let this be equation A
$2S(n) = 1.2^2 + 2.2^3 + ... + (n-1)2^n + n2^{n+1}$ –let this be equation B
$B - A = S(n) = (-1).(2 + 2^2 + 2^3 + ... + 2^n) + n2^{n+1}$

$(2 + 2^2 + 2^3 + ... + 2^n) = \frac{2(2^n-1)}{2-1} = 2^{n+1} - 2$

Substituting this in equation for $S(n)$
$S(n) = n2^{n+1} - 2^{n+1} + 2 = 2^{n+1}(n-1) + 2$
Therefore:
$f(n) = f(0) + S(n)$
Given $f(0) = 3$
$f(n) = 3 + 2^{n+1}(n-1) + 2$

Answer: $f(n) = 2^{n+1}(n-1) + 5$

■

b. $f(n) = 2f(n-1) + 1,\ n > 1$ and $f(1) = 1$.

Solution: You can iteratively replace $f(\cdot)$ on the right using the given recursive equation until you get to the base case. At that point use, the geometric series formula. $f(n) = 2f(n-1) + 1$
$f(n) = 4f(n-2) + 2 + 1$
$f(n) = 8f(n-3) + 4 + 2 + 1$
$f(n) = 2^{n-1}f(n-(n-1)) + 2^{n-2} + ... + 4 + 2 + 1$
$f(n) = 2^{n-1}f(1) + 2^{n-2} + ... + 4 + 2 + 1$
Given $f(1) = 1$
$f(n) = 2^{n-1} + 2^{n-2} + ... + 4 + 2 + 1$
$f(n) = 2^0 + 2^1 + 2^2 + 2^{n-2} + 2^{n-1}$

Using formula $\sum_{k=0}^{n} ar^k = \frac{a(1-r^{n+1})}{1-r}$
where a = 1 and r = 2

$f(n) = \frac{1(1-2^{(n-1)+1})}{1-2}$

Answer: $f(n) = 2^n - 1$

■

c. $f(n) = 2f(\frac{n}{2}) + n,\ n > 1$ and $f(1) = 1$.

Solution: You can iteratively replace $f(\cdot)$ on the right using the given recursive equation until you get to the base case. At that point use, the geometric series formula. $f(n) = 2f(\frac{n}{2}) + n$
$f(n) = 2(2f(\frac{n}{4}) + \frac{n}{2}) + n$
$f(n) = 4f(\frac{n}{4}) + n + n$

At every iteration, the value of n gets halved, so
$f(n) = nf(\frac{n}{n}) + n\log_2 n$
$f(n) = nf(1) + n\log_2 n$
Given $f(1) = 1$

Answer: $f(n) = n(1 + \log_2 n)$

■

Problem 3 [20 points]

The longest palindromic subsequence (LPS) of a sequence is defined as a subsequence of maximum length that is also a palindrome. For example, given the sequence `BANANA`, an LPS is `ANANA` and has length 5.

Write a dynamic programming algorithm to obtain the length of an LPS of a given sequence by providing the following.

- **Recurrence and short English description (in terms of the parameters):**

  Solution: Refer to Lab 10. Alternate solution: LPS = LCS(sequence, reverse-sequence). Also read this. The recurrence is given as follows.

$$LPS(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \max\begin{cases} LPS(i+1,j) \\ LPS(i,j-1) \end{cases} & \text{if } i < j \text{ and } A[i] \neq A[j] \\ 2 + LPS(i+1,j-1) & \text{if } i < j \text{ and } A[i] = A[j] \end{cases}$$

  $LPS(i,j)$ is the length of the longest palindromic subsequence of $A[i..j]$. ■

- **Memoization data structure and evaluation order:**

  Solution: We could use a 2-dimensional $n \times n$ array for memorization. We evaluate in decreasing i, increasing j order. ■

- **Return value:**

  Solution: $LPS[1,n]$ ■

- **Time Complexity:**

  Solution: $O(n^2)$, because we must fill out $O(n^2)$ cells and filling out each cell takes $O(1)$. ■

Problem 4 [20 points]

Consider the graph in Figure 2.

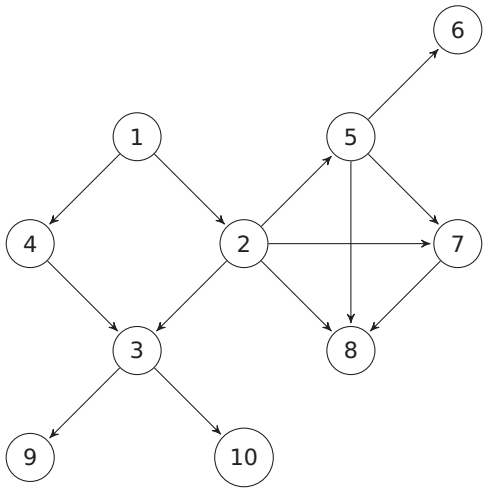


Figure 1. Graph.

Perform DFS starting from vertex 1 while breaking ties in the numeric order, i.e., the node with a smaller numeric label is visited first in tying situations, and answer the following.

a. What is the DFS traversal order, i.e., the order in which you visit different vertices?

Solution: It's the pre-ordering of traversal: 1, 2, 3, 9, 10, 5, 6, 7, 8, 4. ■

b. What are the DFS pre- and post-numberings of different vertices?

Solution: See Figure 2 for the '[pre-number, post-number]' for each node.

Solution: Refer to Lecture 15 and 16 notes, scribbles, and videos. ■

Table 2.

Node	1	2	3	4	5	6	7	8	9	10
Pre numbering	1	2	3	18	9	10	12	13	4	6
Post numbering	20	17	8	19	16	11	15	14	5	7

c. Draw the DFS spanning tree.

Solution: See Figure 3.

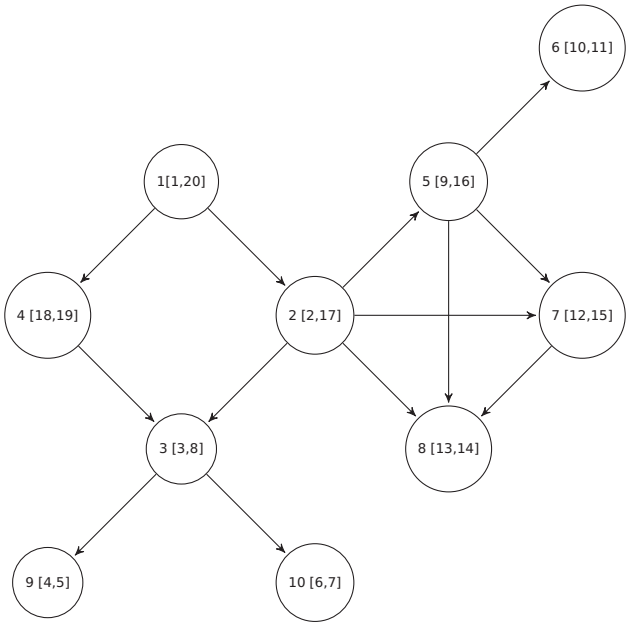


Figure 2. Graph with DFS pre and post-numberings.

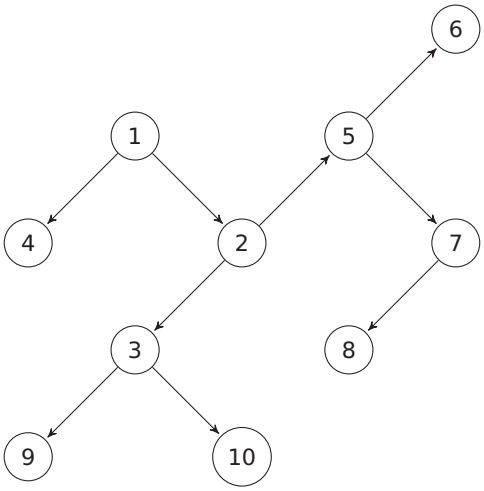


Figure 3. DFS Spanning Tree.

■

d. Classify the non-tree edges into forward, backward, and cross edges. You may draw these non-tree edges in the DFS spanning tree using different edge styles, colors, etc.

Solution: Cross edge: $4 \to 3$. Forward edges: $2 \to 8$, $2 \to 7$, and $5 \to 8$.

■

e. Using DFS or otherwise, obtain a topological sort of the given graph.

Solution: The topological order is 1, (2, 4), (3, 5), (9, 10, 6, 7), 8. The answer is not unique as the nodes in the same parentheses may exchange order. ■

Problem 5 [20 points]

a. Consider the following recursive equation.

$$A(u,v) = \sum_{i=1}^{\min(u,v)} u\,A(u+v-i, i-1), \;\; A(0,v) = v, \text{ and } A(u,0) = u.$$

Analyze the runtime of the memoized implementation of the above recursion to compute $A(n,n)$. Show your work.

Solution: Refer to Lecture 13 Pre-lecture brain teaser.

There are $O(n^2)$ subproblems given two input parameters, i.e., u and v, and each subproblem takes $O(n)$ to solve due to the summation. Therefore, it takes $O(n^2) \cdot O(n) = \boxed{O(n^3)}$ time to compute $A(n,n)$.

■

b. What is the relation between edit distance and longest common subsequence? Explain in detail.

Solution: Refer to Lecture 13 Slide 31 titled Longest common subsequence is just edit distance for the two sequences. Also, refer to the relation between edit distance and alignment in the same lecture.

- The longest-increasing subsequence problem asks for the length of the longest-increasing subsequence in an unordered sequence, where the sequence is assumed to be given as an array.
- The edit distance problem asks how many edits we need to make to a sequence for it to become another one.
- Both are alignment problems. The costs can be written differently for each of the problems. To solve LCS using the concept of ED, we need to make two modifications.
  - For letters that are different in the two strings, change the cost of replacing a character with the correct character to ∞. Since we are minimizing the ED, it is never viable at all times to replace the letters if they are different.
  - For letters that are the same, the cost of replacing them is 1.
  - The cost of insertion and deletion are still 1.
- Satisfying the above modifications to ED, LCS and ED are complements, and the relation between them is LCS(A, B) = len(A) + len(B) - ED(A, B).

■

c. Analyze the runtime of the median-of-median algorithm with the group size as 3 instead of the usual 5.

Solution: Refer to Lecture 11 Pre-lecture brain teaser. 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 $|A_{\text{less}}|$ 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 $O(n \log(n))$. ■

d. Provide a logarithmic time algorithm to count the number of instances of a given number in a sorted list.

Solution: We can slightly modify binary search to find the leftmost array element that contains x (the left-bound of the array block):

```
FindLeftBound(A[1..n], x, i):
  if A[1] = x
      return i
  else
      if A[n/2] ≥ x
          return FindLeftBound(A[1, ..., n/2], x, i)
      else
          return FindLeftBound(A[n/2 + 1, ..., n], x, i + n/2)
```

i is a variable to keep track of the original position of the sub-array being currently evaluated. We do the same to find the right bound and subtract the two values from one another to find the number of instances of x as we are using modified binary search twice, we have log runtime. ■

Problem 6 [15 points]

Consider the problem of multiplying n, m-digit numbers, $m \geq n$. One simple strategy to solve this problem is to use Karatsuba's algorithm $n-1$ times, i.e., multiply the first number with the second number, then their product with the third number, and so on. Answer the following.

a. Analyze the runtime of the above simple strategy.

Solution: When multiplying 2 "m-digit" numbers, the upper bound on product size is $2m = O(m)$.
Example: 99 (2 digits) × 999 (3 digits) = 98901 (5 digits).

Take array $A = [a_1, a_2, a_3, ..., a_{n-1}, a_n]$. The product size of different prefixes is:

(a) $[a_1, a_2] \rightarrow$ product size $\approx 2m$
(b) $[a_1, a_2, a_3] \rightarrow$ product size $\approx 3m$
(c) $[a_1, a_2, a_3, a_4] \rightarrow$ product size $\approx 4m$
(d) $[a_1, a_2, ..., a_n] \rightarrow$ product size $\approx nm$

Therefore, the running time is

$$\begin{aligned}\sum_{k=1}^{n}(km)^{\log 3} &= (m)^{\log 3} + (2m)^{\log 3} + (3m)^{\log 3} + ... + (nm)^{\log 3} \\ &\leq O(n) \times O((nm)^{\log 3}) && \left[(km)^{\log 3} \leq O((nm)^{\log 3} \forall k\right] \\ &\leq \boxed{O(n(nm)^{\log 3})}\end{aligned}$$

■

b. Provide a more time-efficient divide-and-conquer recursive algorithm to solve the given problem.

Solution: The input is $A[1..n]$, a list of m-digit numbers:

```
FastProd(A[1..n]):          ⟨⟨T(n)⟩⟩
  if n = 1
      return A[1]
  p1 ← FastProd(A[1..n/2])        ⟨⟨T(n/2)⟩⟩
  p2 ← FastProd(A[n/2+1..n])      ⟨⟨T(n/2)⟩⟩
  ⟨⟨Running time of Karatsuba on 2 k-digit numbers is O(k^log3)⟩⟩
  ⟨⟨Max size p1, p2 is nm/2⟩⟩
  return Karatsuba(p1, p2)        ⟨⟨So, O((nm)^log3)⟩⟩
```

■

c. Analyze the runtime of your algorithm.

Solution: We model this recurrence as a recursion tree.
Recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + O\left((nm)^{\log 3}\right)$

$$(nm)^{\log 3}$$

$$\left(\frac{n}{2}m\right)^{\log 3} \qquad \left(\frac{n}{2}m\right)^{\log 3}$$

$$\left(\frac{n}{4}m\right)^{\log 3} \quad \left(\frac{n}{4}m\right)^{\log 3} \quad \left(\frac{n}{4}m\right)^{\log 3} \quad \left(\frac{n}{4}m\right)^{\log 3}$$

The amount of work at each level decreases due to the logarithm therefore dominated by the root, so the recurrence is $\boxed{O((nm)^{\log 3})}$. ■

This page is for additional scratch work!

ECE 374 B Algorithms: Cheatsheet

1 Recursion

Simple recursion

- **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).

Definitions

- 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*

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.

Tower of Hanoi

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)
```

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	$O(n \log n)$	$O(n \log n)$ $O(n)$ (if optimized)
Quicksort	$O(n^2)$ $O(n \log n)$ if using MoM	$O(n)$

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:

Karatsuba's algorithm

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

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

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: $\log(ab) = \log a + \log b, \log(a/b) = \log a - \log b, a^{\log_c b} = b^{\log_c a}$ $(a, b, c > 1)$.

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 ← 0, 5, . . . , len(A)]
  medians = [sorted (sublist)[len (sublist)/2]
             for sublist ∈ sublists]

  // Base case
  if len (A) ≤ 5 return sorted (a)[i]

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

  // Partitioning step
  low = [j for j ∈ A  if j < pivot]
  high = [j for j ∈ 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
```

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
```

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:

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

Pseudocode: LIS - DP

```
LIS-Iterative(A[1..n]):
    A[n+1] = ∞
    for j ← 0 to n
        if A[i] ≤ A[j] then LIS[0][j] = 1

    for i ← 1 to n − 1 do
        for j ← i to n − 1 do
            if A[i] ≥ 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 ← 1 to m do M[i, 0] = iδ
    for j ← 1 to n do M[0, j] = jδ

    for i = 1 to m do
        for j = 1 to n do
            M[i][j] = min { COST[A[i]][B[j]]
                            +M[i − 1][j − 1],
                          δ + M[i − 1][j],
                          δ + M[i][j − 1]
```

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 algorithm

Kahn$(G(V, E), u)$:
toposort$\leftarrow$empty list
for $v \in V$:
  in$(v) \leftarrow |\{u \mid u \to v \in E\}|$
while $v \in V$ that has in$(v) = 0$:
  Add v to end of toposort
  Remove v from V
  for v in $u \to v \in E$:
    in$(v) \leftarrow$ in$(v) - 1$
return toposort

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):
  for $i \leftarrow 1$ to n:
    Visited$[i] \leftarrow$ False
  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, S, T (with x as parent)

Note:

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

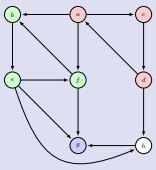
Pre/post numbering

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:

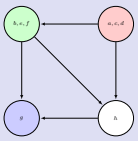
- *Forward edge*: pre(u) < pre(v) < post(v) < post(u)
- *Backward edge*: pre(v) < pre(u) < post(u) < post(v)
- *Cross edge*: pre(u) < post(u) < pre(v) < post(v)

Strongly connected components

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



G^{SCC}



Pseudocode: Metagraph - linear time

Metagraph$(G(V, E))$:
  Compute rev(G) by brute force
  ordering $\leftarrow$ reverse postordering of V in rev(G)
    by **DFS**(rev$(G), s)$ for any vertex s
  Mark all nodes as unvisited
  for each u in ordering **do**
    if u is not visited and $u \in V$ **then**
      $S_u \leftarrow$ nodes reachable by u by **DFS**(G, u)
      Output S_u as a strong connected component
      $G(V, E) \leftarrow G - S_u$

Shortest paths

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

Pseudocode: Dijkstra

for $v \in V$ **do**
  $d(v) \leftarrow \infty$
$X \leftarrow \varnothing$
$d(s, s) \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $v \leftarrow \arg\min_{u \in V - X} d(u)$
  $X = X \cup \{v\}$
  for u in Adj(v) **do**
    $d(u) \leftarrow \min\{(d(u),\ d(v) + \ell(v, u))\}$
return d

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:

$$d(v, k) = \begin{cases} 0 & \text{if } v = s \text{ and } k = 0 \\ \infty & \text{if } v \neq s \text{ and } k = 0 \\ \min \begin{cases} \min_{uv \in E} \{d(u, k-1) + \ell(u, v)\} \\ d(v, k-1) \end{cases} & \text{else} \end{cases}$$

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

Pseudocode: Bellman-Ford

for each $v \in V$ **do**
  $d(v) \leftarrow \infty$
$d(s) \leftarrow 0$

for $k \leftarrow 1$ to $n - 1$ **do**
  for each $v \in V$ **do**
    for each edge $(u, v) \in$ in(v) **do**
      $d(v) \leftarrow \min\{d(v), d(u) + \ell(u, v)\}$

return d

Running time: $O(nm)$

Floyd-Warshall algorithm:
Find minimum distance from *every* vertex to *every* vertex in a graph *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(i, j, k) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (i, j) \notin E \text{ and } k = 0 \\ \min \begin{cases} d(i, j, k-1) \\ d(i, k, k-1) + d(k, j, k-1) \end{cases} & \text{else} \end{cases}$$

Then $d(i, j, n - 1)$ will give the shortest-path distance *from* i *to* j.

Pseudocode: Floyd-Warshall

Metagraph$(G(V, E))$:
  for $i \in V$ **do**
    for $j \in V$ **do**
      $d(i, j, 0) \leftarrow \ell(i, j)$
        `(* ℓ(i,j) ← ∞ if (i,j) ∉ E, 0 if i = j *)`

  for $k \leftarrow 0$ to $n - 1$ **do**
    for $i \in V$ **do**
      for $j \in V$ **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 $v \in V$ **do**
    if $d(i, i, n - 1) < 0$ **then**
      return "$\exists$ negative cycle in G"

  return $d(\cdot, \cdot, n - 1)$

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