CSE 421 Winter 2026: Section 1

January 8, 2026

Instructions: This section worksheet is designed to assist you in working through some example problems and developing your basics. You are encouraged to collaborate on the problems on this section as well as use the course's generative AI.

1 Introduction

Welcome to CSE 421! The purpose of CSE 421 Sections is to help you become comfortable with the basics and give you a chance to ask questions one-on-one with the TAs in this course. This first Section will provide a refresher on Big-O notation and help you become comfortable with writing algorithms and proofs of correctness. During your section, your TA will work with you through some of the problem but will almost certainly not solve all the problems on the worksheet. You can use the rest of the worksheet as practice, however you see fit.

Section worksheets include solutions for all the problems written in the level of specificity we prefer at the end of the worksheet. This is designed to help you practice writing solutions concisely and accurately. Remember, this is a course emphasizing the theory of algorithms. We are interested in not only constructing efficient algorithms but also proving that the algorithms are correct. On the course webpage, there are instructions and suggestions on how to write algorithms. This first section provides examples. As you go through today's section, observe that none of the algorithms are written in pseudocode. Instead, we write the algorithms in *plain technical English.* A programmer could then take the algorithm and produce code in any programming language.

2 Big-O notation

The following questions are to help you practice Big-O notation skills.

2.1 Warm-Up Calculations (Big-O)

For each pair of functions $f(n), g(n)$, determine whether $f(n) = O(g(n))$, $g(n) = O(f(n))$, **both**, or **neither**. Briefly justify your answer.

1. $f(n) = 3n^2 + 7n + 4, \quad g(n) = n^2$

2. $f(n) = n \log n, \quad g(n) = n^{1.1}$
3. $f(n) = \log(n!), \quad g(n) = n \log n$
4. $f(n) = 2^n, \quad g(n) = n!$
5. $f(n) = \log \log n, \quad g(n) = \log n$

2.2 Big-Ω Practice

For each statement below, determine whether it is **true or false**. If true, give a short proof; if false, give a counterexample.

1. If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.
2. If $f(n) = \Omega(g(n))$, then $g(n) = O(f(n))$.
3. If $f(n) = \Omega(g(n))$, then $f(n) + h(n) = \Omega(g(n))$ for all $h(n) \geq 0$.

2.3 Quantifiers Matter

Decide whether each statement is true or false. Your justification must explicitly reference the definition of Big-O or Big-Ω.

1. If $f(n) = O(g(n))$, then there exists a constant $c > 0$ such that $f(n) \leq cg(n)$ for *all* $n \geq 1$.
2. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
3. If $f(n) = O(g(n))$, then $f(n)^2 = O(g(n)^2)$.

2.4 Comparing Exponential-Like Functions

Order the following functions from **slowest** to **fastest** growing. You do not need to give a formal proof, but you must give a convincing argument.

$$n^{\log n}, \quad 2^{\sqrt{\log n}}, \quad (\log n)^{\log n}, \quad 2^{(\log n)^2}$$

3 Induction practice

In past courses, you have been introduced to proofs by induction. Proofs by induction take various forms and now that you are in CSE 421, we will let you be less formal in your writing of a proof while still maintaining rigor. Take a look at our solutions after you practice solving them on your own – notice how we are more casual with writing out the induction. Also, all the following examples are proofs by induction even if they don't seem to be at first glance!

3.1 A Simple Graph Induction Problem

Let $G = (V, E)$ be a connected undirected graph with $n \geq 2$ vertices. Suppose that G has exactly $n - 1$ edges.

1. Prove that G contains no cycles.
2. Prove that between every pair of vertices in G, there is a unique simple path.

Definitions and remarks. A *simple path* is a path that does not repeat vertices. A *cycle* is a simple path that starts and ends at the same vertex. A graph that is connected and contains no cycles is called a *tree*. This problem shows that the edge-count condition $|E| = |V| - 1$ forces a tree structure.

3.2 Induction by Minimal Counterexample

Prove that every integer $n \geq 2$ can be written as a product of prime numbers.

For this problem, practice writing your solution as a "proof by contradiction". Consider the minimal example n for which the statement is false.

3.3 A Flawed Induction with Big-O

Consider the recurrence $T(n) = T(n-1) + n$ with base case $T(1) = 1$. Little Johnny attempts to prove by induction that $T(n) = O(n^2)$ as follows.

> *Inductive hypothesis:* Assume that for some constant $c > 0$, $T(n-1) \leq c(n-1)^2$. Then,
>
> $$T(n) = T(n-1) + n \leq c(n-1)^2 + n \leq cn^2 + n.$$
>
> Since $cn^2 + n \leq (c+1)n^2$ for all $n \geq 1$, we conclude that $T(n) \leq (c+1)n^2$, so $T(n) = O(n^2)$.

1. Identify the step(s) where the inductive reasoning fails.
2. Give a correct argument that $T(n) = O(n^2)$.

3.4 Balanced Parentheses

Given a string S consisting only of the characters '(' and ')', describe an algorithm that determines whether S is *balanced*. A string is balanced if every opening parenthesis is matched with a later closing parenthesis and the parentheses are properly nested.
Practice describing your algorithm in clear, precise technical English and prove (using induction) that your algorithm is correct.

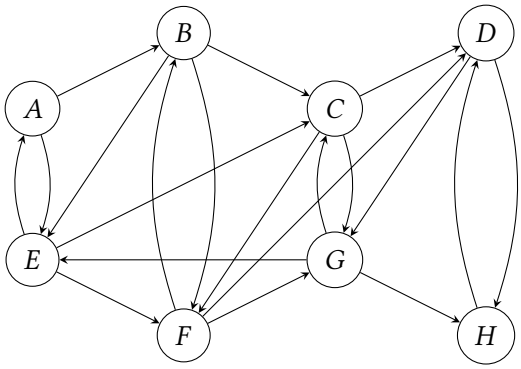
Definitions and remarks. A prefix of a string S is any substring of the form $S[1..k]$ for some $k \geq 0$. A string of parentheses is balanced if and only if every prefix contains at least as many opening parentheses as closing parentheses, and the total number of opening and closing parentheses in the string are equal. This problem is a simple example of designing an algorithm and proving its correctness using induction on the length of the input.

4 Graphs

4.1 Practice BFS and DFS

Practice running BFS and DFS on the following graph from A and computing the enumeration of the vertices by visit order (similar to that in the lecture). Assume the graph traversal adds neighbors to the stack or queue in alphabetical order.

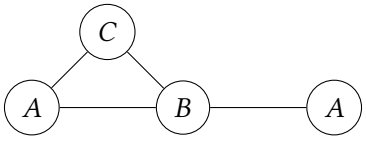
Additionally, you can practice computing the BFS and DFS search trees.



4.2 Coloring a graph

Given a connected undirected graph $G = (V, E)$ with $n = |V|$ vertices and n edges. Construct an $O(n)$ time algorithm to color the vertices of G with 3 colors such that any two adjacent pair of vertices have distinct colors. Prove runtime and correctness.

For example, in the following graph $n = 4$ and we have colored the vertices with 3 colors: A, B, C.



Definitions and remarks. A coloring of a graph with k colors is an assignment of a color $\{1, \ldots, k\}$ to each vertex in the graph with the property that any neighboring vertices are assigned different colors. While

the problem of deciding if a graph G can be colored with 3 colors is NP-complete (in general), this is a special case which can be solved efficiently.

This problem is a great practice for recognizing a property in a problem and then finding an algorithm that exploits the property.

Hints:

1. Prove that the average degree of vertices in this graph is 2.
2. Argue that either all the vertices have degree 2 or at least one vertex has degree 1.
3. If all the vertices have degree 2 then what does the graph look like? (Answer: union of cycles)
4. If a vertex has degree 1, then you should be able to recurse.

5 Solutions

5.2 Big-O

5.2.1 Solution to Warm-Up Calculations (Big-O)

1. **Both.** Polynomial lower-order terms do not affect asymptotic growth, so $3n^2 + 7n + 4 = \Theta(n^2)$.
2. $f(n) = O(g(n))$**.** Since $\log n = o(n^{0.1})$, we have $n \log n = o(n^{1.1})$.
3. **Both.** By Stirling's approximation, $\log(n!) = \Theta(n \log n)$.
4. $f(n) = O(g(n))$**.** Factorial growth dominates exponential growth: $2^n = o(n!)$.
5. $f(n) = O(g(n))$**.** $\log \log n$ grows strictly slower than $\log n$.

5.2.2 Solution to Big-Ω Practice

1. **True.** This follows directly from transitivity of asymptotic dominance.
2. **True.** By definition, $f(n) = \Omega(g(n))$ means $g(n) = O(f(n))$.
3. **True.** Since $f(n) \geq cg(n)$ eventually and $h(n) \geq 0$, we have $f(n) + h(n) \geq cg(n)$.

5.2.3 Solution to Quantifiers Matter

1. **False.** Big-O only requires the inequality to hold for sufficiently large n, not all n.
2. **True.** Big-O is transitive by composition of constants and thresholds.
3. **True.** Squaring preserves asymptotic dominance when functions are eventually nonnegative.

5.2.4 Solution to Comparing Exponential-Like Functions

Rewriting the functions:

$$(\log n)^{\log n} = 2^{(\log n)(\log \log n)}, \quad n^{\log n} = 2^{(\log n)^2}$$

Thus the growth order is:

$$2^{\sqrt{\log n}} < (\log n)^{\log n} < n^{\log n} = 2^{(\log n)^2}$$

So from slowest to fastest:

$$2^{\sqrt{\log n}}, \quad (\log n)^{\log n}, \quad n^{\log n}, \quad 2^{(\log n)^2}.$$

5.3 Induction practice

5.3.1 Simply connected graph

We prove the claim by induction on the number of vertices n. For the base case $n = 2$, the graph has exactly one edge, so it is connected, contains no cycles, and the unique path property is immediate. Now assume the claim holds for all connected graphs on $n-1$ vertices with exactly $n-2$ edges, and let G be a connected graph on n vertices with $n-1$ edges. The average degree of G is $\frac{2(n-1)}{n} = 2 - \frac{2}{n} < 2$, and since degrees are integers and G is connected, there must exist a vertex v of degree exactly 1. Removing v and its incident edge (v, v') yields a connected graph G' with $n-1$ vertices and $n-2$ edges. By the inductive hypothesis, G' contains no cycles and has a unique simple path between every pair of vertices. Adding v back into the graph cannot create a cycle as v has degree 1 and every vertex on a cycle has degree ≥ 2. Secondly, it introduces exactly one simple path from v to every other vertex w as the edge (v, v') must be included and there is a unique simple path from $v' \rightsquigarrow w$.

5.3.2 Prime divisors

Suppose, for contradiction, that the statement is false, and let $n \geq 2$ be the smallest integer that cannot be written as a product of primes[1]. The number n cannot itself be prime, since a prime is trivially a product of a single prime. Therefore, n is composite, so there exist integers a, b with $2 \leq a, b < n$ such that $n = ab$. By the minimality of n, both a and b can be written as products of primes. Multiplying these prime factorizations yields a prime factorization of n, contradicting the assumption that n is a counterexample. Thus, no such counterexample exists, and every integer $n \geq 2$ can be written as a product of prime numbers.

5.3.3 Incorrect induction proof

The error in the proof is subtle but fundamental: the inductive hypothesis assumes the existence of a fixed constant c, but the inductive step replaces it with a larger constant $c+1$. This means the constant depends on the number of induction steps taken, and thus implicitly depends on n. Big-O notation requires the existence of a *single fixed constant* that works for all sufficiently large n, not a sequence of constants that grows with n.

The inductive reasoning fails at the step where Little Johnny concludes that $cn^2 + n \leq (c+1)n^2$ and then reuses $(c+1)$ as the constant in the next inductive step. Each iteration increases the constant, so the proof does not establish a uniform upper bound of the form $T(n) \leq Cn^2$ for a fixed C.

To give a correct proof, we instead choose the constant *in advance*. We claim that $T(n) \leq \frac{1}{2}n^2 + \frac{1}{2}n$ for all $n \geq 1$. This can be verified by induction using the recurrence:

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

[1]Considering (for the purposes of contradiction), the smallest instance which doesn't follow the statement is a rewording of the standard inductive proof.

Thus $T(n) = O(n^2)$ with an explicit constant bound.

There is also a second mistake that $T(1)$ is not less than $c(1-1)^2$ for any constant c, and therefore the base case is not valid.

5.3.4 Balanced parantheses

Algorithm The algorithm scans the string from left to right, maintaining a counter equal to the number of opening parentheses that have not yet been matched. The counter is incremented on reading '(' and decremented on reading ')'. If the counter ever becomes negative, the algorithm reports "not balanced." After the scan finishes, the algorithm reports "balanced" if and only if the counter is zero.

Correctness We prove correctness by induction on the length of the input string. For a string of length 0, the string is empty and hence balanced. The algorithm performs no operations, the counter remains 0, and it correctly reports "balanced." Assume the algorithm is correct for all strings of length less than n, and let S be a string of length n.

If during the scan the counter ever becomes negative, then some prefix of S contains more closing parentheses than opening parentheses, so S cannot be balanced and the algorithm correctly reports "not balanced." Otherwise, the scan completes without the counter becoming negative. If the final counter value is nonzero, then S contains more opening parentheses than closing parentheses and cannot be balanced, so the algorithm again correctly reports "not balanced."

Finally, suppose the scan completes with final counter value 0. Let j be the first position at which the counter returns to 0. Then the prefix $S[1..j]$ begins with '(', ends with ')', and no proper prefix of it causes the counter to become negative, so $S[1..j]$ is balanced. The remaining suffix $S[j+1..n]$ has length less than n and also satisfies the balance condition. By the inductive hypothesis, the algorithm correctly recognizes the suffix as balanced. Therefore, S is a concatenation of balanced strings and is itself balanced, and the algorithm correctly reports "balanced." This completes the induction and proves correctness.

Runtime. The algorithm examines each character once and performs constant work per character, so it runs in $O(n)$ time.

5.4 Graph algorithms

5.4.1 Practice BFS and DFS

The correct BFS enumeration is (1) A, (2) B, (3) E, (4) C, (5) F, (6) D, (7) G, (8) H.

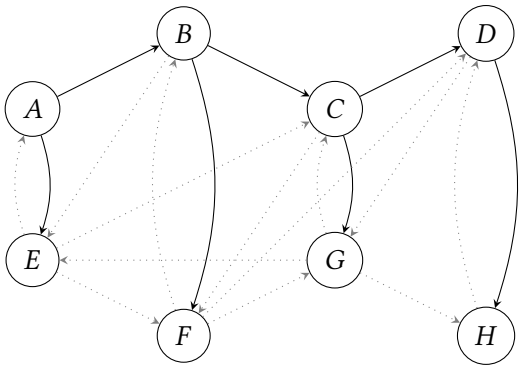


Figure 1: BFS tree

The correct DFS enumeration is (1) A, (2) E, (3) F, (4) G, (5) H, (6) D, (7) C, (8) B.

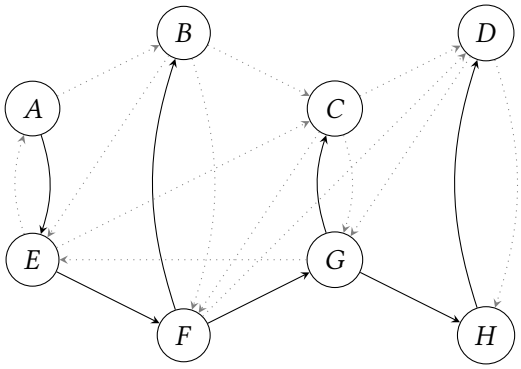


Figure 2: DFS tree

5.4.2 3-coloring a graph

The key step for this algorithm is the observation that a graph with n vertices and n edges must either contain a degree-1 vertex or be a union of cycles. This is because the average degree is 2. Therefore, the minimum degree is either 1 or 2 as the minimum degree is $\leq$ the average degree. If the minimum degree is 2, then the maximum degree must also be 2 for the average degree to be 2. And a graph with degree 2 is the union of cycles.

Algorithm. Compute the degree $\deg(v)$ of every vertex and initialize a stack S. Initialize a queue Q with all the degree-1 vertices. While the queue Q is non-empty with top element v with neighbor u, push (v, u) onto S and remove v from the graph. In removal, update degrees and add new degree-1 vertices to Q.

Once Q is empty, the remaining connected graph has all degrees 2, hence it is a cycle; traverse this

cycle and color its vertices $1, 2, 1, 2, 1, 2, \ldots$ in order with the last vertex potentially colored 3. Finally, pop pairs (v, u) from S in reverse removal order and assign v any color in $\{1, 2, 3\}$ not equal to the color of u.

Correctness. We prove that the algorithm outputs a proper 3-coloring. First consider the stage when the while-loop ends. At that moment, either the remaining graph is empty or every remaining vertex has degree 2. Because the graph was connected and removing leaves never disconnects the remaining vertices from each other, the remaining graph (if nonempty) is connected with all degrees 2, hence it is a single cycle. Coloring a cycle with the repeating pattern $1, 2, 3, 1, 2, 3, \ldots$ is proper because adjacent vertices on the cycle always get different colors.

Now consider any vertex v popped from the stack with stored neighbor u. At the moment we color v, the vertex u is already colored (since u remained in the graph when v was removed). We assign v a color different from $\text{color}(u)$, which is always possible with 3 colors. Thus the edge (v, u) is properly colored. Moreover, v had degree 1 at the time it was removed, so in the original graph its only neighbor among the not-yet-removed vertices was u; therefore ensuring $\text{color}(v) \neq \text{color}(u)$ suffices to make all edges incident to v properly colored. By processing all popped vertices, every edge in G ends up properly colored, so the final coloring is a valid 3-coloring.

Runtime. We maintain degrees and process each vertex at most once (pushed/popped once) and each edge a constant number of times when updating degrees. Since $|E| = |V| = n$, the total running time is $O(|V| + |E|) = O(n)$.