UCLA COMPUTER SCIENCE FIELD NOTES

CS 180

Introduction to Algorithms and Complexity

Proof, divide-and-conquer, greedy choice, dynamic programming, graph algorithms, flow, and NP-completeness

How these notes are written. I wrote each chapter the way I wish the material had been explained the first time: the real idea, the point where I usually get stuck, the simplest mental model that still stays honest, and a worked decision instead of a polished answer appearing from nowhere. Every chapter closes with practice and a fully written solution. If a sentence sounds impressive but does not help me solve or explain something, it does not belong here.

Daniel Mastick

Curriculum map, working notes, practice, and solutions

Course map

Week	Core thread	What should be solid by the end
1	**Correctness and asymptotic reasoning**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
2	**Divide-and-conquer**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
3	**Greedy algorithms**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
4	**Dynamic programming**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
5	**Graph traversal and DAG algorithms**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
6	**Shortest paths and minimum spanning trees**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
7	**Network flow and reductions**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
8	**NP-completeness and the limits of efficient search**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
9	**Integration workshop**	Connect at least three chapters in one end-to-end problem and explain where their contracts meet.
10	**Synthesis and project review**	Audit assumptions and explain a complete solution from interface to failure handling.

Scope anchor: https://catalog.registrar.ucla.edu/course/2025/COMSCI180. The sequence is aligned to the official catalog description and expanded into a practical ten-week study plan.

How I would use this packet

Treat each numbered section as one chapter. Read the formal explanation, pause at the student interpretation, and see whether the easy version still says the same thing. Rework the example without looking, then cover the solution in the chapter practice box. The cumulative set at the end is deliberately mixed: knowledge becomes durable when I have to choose the tool instead of being told which chapter I am in.

Contents

1 Correctness and asymptotic reasoning **5**

1.1 Rebuild the chapter from first principles . . . 5
1.2 Details I would refuse to skip . . . 5
1.3 How this chapter connects . . . 6
1.4 What I need to be able to do . . . 6
1.5 Deep notes . . . 6

2 Divide-and-conquer **7**

2.1 Rebuild the chapter from first principles . . . 7
2.2 Details I would refuse to skip . . . 8
2.3 How this chapter connects . . . 8
2.4 What I need to be able to do . . . 8
2.5 Deep notes . . . 9

3 Greedy algorithms **10**

3.1 Rebuild the chapter from first principles . . . 10
3.2 Details I would refuse to skip . . . 11
3.3 How this chapter connects . . . 11
3.4 What I need to be able to do . . . 11
3.5 Deep notes . . . 11

4 Dynamic programming **13**

4.1 Rebuild the chapter from first principles . . . 13
4.2 Details I would refuse to skip . . . 13
4.3 How this chapter connects . . . 14
4.4 What I need to be able to do . . . 14
4.5 Deep notes . . . 14

5 Graph traversal and DAG algorithms **15**

5.1 Rebuild the chapter from first principles . . . 15
5.2 Details I would refuse to skip . . . 16
5.3 How this chapter connects . . . 16
5.4 What I need to be able to do . . . 16
5.5 Deep notes . . . 17

6 Shortest paths and minimum spanning trees **18**

6.1 Rebuild the chapter from first principles . . . 18

6.2 Details I would refuse to skip . . . 19

6.3 How this chapter connects . . . 19

6.4 What I need to be able to do . . . 19

6.5 Deep notes . . . 19

7 Network flow and reductions **21**

7.1 Rebuild the chapter from first principles . . . 21

7.2 Details I would refuse to skip . . . 21

7.3 How this chapter connects . . . 22

7.4 What I need to be able to do . . . 22

7.5 Deep notes . . . 22

8 NP-completeness and the limits of efficient search **23**

8.1 Rebuild the chapter from first principles . . . 23

8.2 Details I would refuse to skip . . . 24

8.3 How this chapter connects . . . 24

8.4 What I need to be able to do . . . 25

8.5 Deep notes . . . 25

CHAPTER 1 Correctness and asymptotic reasoning

The idea

Algorithm analysis begins with a precise problem, model, invariant, and proof. Upper and lower bounds distinguish an implementation guarantee from an inherent problem cost.

Rebuild the chapter from first principles

The claim I need to own is this: Algorithm analysis begins with a precise problem, model, invariant, and proof. Upper and lower bounds distinguish an implementation guarantee from an inherent problem cost. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of correctness and asymptotic reasoning. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach correctness and asymptotic reasoning on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from the course foundations to divide-and-conquer. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Algorithm analysis begins with a precise problem, model, invariant, and proof.
- **Mechanism.** Upper and lower bounds distinguish an implementation guarantee from an inherent problem cost.
- **Boundary.** The useful test is whether I can apply correctness and asymptotic reasoning to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when correctness and asymptotic reasoning is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I separate three claims: the algorithm terminates, returns a legal answer, and returns the best answer.

The easy version

Algorithm analysis begins with a precise problem, model, invariant, and proof.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Prove insertion sort's outer-loop invariant. The conclusion is Before iteration i, the prefix 0 through i-1 is sorted and is a permutation of the original prefix. Inserting a[i] preserves both facts; at i=n the full array is sorted.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Prove insertion sort's outer-loop invariant.
Solution. Before iteration i, the prefix 0 through i-1 is sorted and is a permutation of the original prefix. Inserting a[i] preserves both facts; at i=n the full array is sorted.

Practice 2. Give a short oral explanation of correctness and asymptotic reasoning that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct correctness and asymptotic reasoning from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a correctness and asymptotic reasoning problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 2 Divide-and-conquer

The idea

Divide-and-conquer splits independent subproblems, solves them recursively, and combines results. Recurrences expose whether the combine work, number of branches, or leaf count dominates.

Rebuild the chapter from first principles

The claim I need to own is this: Divide-and-conquer splits independent subproblems, solves them recursively, and combines results. Recurrences expose whether the combine work, number of branches, or leaf count dominates. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions,

and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of divide-and-conquer. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach divide-and-conquer on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from correctness and asymptotic reasoning to greedy algorithms. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method

valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Divide-and-conquer splits independent subproblems, solves them recursively, and combines results.
- **Mechanism.** Recurrences expose whether the combine work, number of branches, or leaf count dominates.
- **Boundary.** The useful test is whether I can apply divide-and-conquer to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when divide-and-conquer is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I draw the recursion tree before applying a theorem; the levels make the cost believable.

The easy version

Divide-and-conquer splits independent subproblems, solves them recursively, and combines results.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Solve T(n)=2T(n/2)+n. The conclusion is There are log n levels and each contributes Theta(n), so T(n)=Theta(n log n). The Master theorem gives the same result.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Solve T(n)=2T(n/2)+n.
Solution. There are log n levels and each contributes Theta(n), so T(n)=Theta(n log n). The Master theorem gives the same result.

Practice 2. Give a short oral explanation of divide-and-conquer that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct divide-and-conquer from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a divide-and-conquer problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 3 Greedy algorithms

The idea

A greedy method commits to a locally best choice. Correctness needs an exchange, stay-ahead, or cut argument showing some optimum can be transformed to include that choice.

Rebuild the chapter from first principles

The claim I need to own is this: A greedy method commits to a locally best choice. Correctness needs an exchange, stay-ahead, or cut argument showing some optimum can be transformed to include that choice. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of greedy algorithms. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach greedy algorithms on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from divide-and-conquer to dynamic programming. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** A greedy method commits to a locally best choice.
- **Mechanism.** Correctness needs an exchange, stay-ahead, or cut argument showing some optimum can be transformed to include that choice.
- **Boundary.** The useful test is whether I can apply greedy algorithms to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when greedy algorithms is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

A greedy rule that feels obvious is only a hypothesis until I can exchange it into an optimal solution.

The easy version

A greedy method commits to a locally best choice.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Prove earliest-finish interval scheduling. The conclusion is Take the earliest-finishing compatible interval. In any optimum replace its first interval with this one; the replacement ends no later and leaves at least as much room, so induction completes the proof.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Prove earliest-finish interval scheduling.
Solution. Take the earliest-finishing compatible interval. In any optimum replace its first interval with this one; the replacement ends no later and leaves at least as much room, so induction completes the proof.

Practice 2. Give a short oral explanation of greedy algorithms that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct greedy algorithms from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a greedy algorithms problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 4 Dynamic programming

The idea

DP applies when subproblems overlap and optimal solutions have compatible substructure. State must contain exactly the past information needed for future decisions; transitions, base cases, evaluation order, and reconstruction complete the design.

Rebuild the chapter from first principles

The claim I need to own is this: DP applies when subproblems overlap and optimal solutions have compatible substructure. State must contain exactly the past information needed for future decisions; transitions, base cases, evaluation order, and reconstruction complete the design. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of dynamic programming. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach dynamic programming on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from greedy algorithms to graph traversal and dag algorithms. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** DP applies when subproblems overlap and optimal solutions have compatible substructure.
- **Mechanism.** State must contain exactly the past information needed for future decisions; transitions, base cases, evaluation order, and reconstruction complete the design.
- **Boundary.** The useful test is whether I can apply dynamic programming to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when dynamic programming is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

My shorthand for this chapter is: my shorthand for this chapter is: most DP difficulty is naming the state, not writing the loop.

The easy version

DP applies when subproblems overlap and optimal solutions have compatible substructure.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Define a state for 0/1 knapsack. The conclusion is DP[i,w] is best value using first i items with capacity w. Exclude item i or include it if weight permits; take the maximum and record choices for reconstruction.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Define a state for 0/1 knapsack.
Solution. DP[i,w] is best value using first i items with capacity w. Exclude item i or include it if weight permits; take the maximum and record choices for reconstruction.

Practice 2. Give a short oral explanation of dynamic programming that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct dynamic programming from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a dynamic programming problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 5 Graph traversal and DAG algorithms

The idea

BFS organizes unweighted distance layers; DFS exposes ancestry, cycles, finishing times, and components. Topological order linearizes a DAG so dependency-constrained dynamic programs can run once per edge.

Rebuild the chapter from first principles

The claim I need to own is this: BFS organizes unweighted distance layers; DFS exposes ancestry, cycles, finishing times, and components. Topological order linearizes a DAG so dependency-constrained dynamic programs can run once per edge. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the

definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of graph traversal and dag algorithms. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach graph traversal and dag algorithms on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from dynamic programming to shortest paths and minimum spanning trees. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method

valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** BFS organizes unweighted distance layers; DFS exposes ancestry, cycles, finishing times, and components.
- **Mechanism.** Topological order linearizes a DAG so dependency-constrained dynamic programs can run once per edge.
- **Boundary.** The useful test is whether I can apply graph traversal and dag algorithms to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when graph traversal and dag algorithms is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I state what the frontier means: BFS frontier is next distance, DFS stack is unfinished ancestry.

The easy version

BFS organizes unweighted distance layers; DFS exposes ancestry, cycles, finishing times, and components.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: How does DFS detect a directed cycle? The conclusion is Color vertices white, gray, black. An edge to a gray ancestor is a back edge and proves a directed cycle; no back edge means the graph is acyclic.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. How does DFS detect a directed cycle?
Solution. Color vertices white, gray, black. An edge to a gray ancestor is a back edge and proves a directed cycle; no back edge means the graph is acyclic.

Practice 2. Give a short oral explanation of graph traversal and dag algorithms that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a

four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct graph traversal and dag algorithms from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a graph traversal and dag algorithms problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 6 Shortest paths and minimum spanning trees

The idea

Relaxation improves a tentative path using one edge. Dijkstra requires nonnegative weights; Bellman-Ford tolerates negatives and detects reachable negative cycles. MST cut and cycle properties justify Kruskal and Prim.

Rebuild the chapter from first principles

The claim I need to own is this: Relaxation improves a tentative path using one edge. Dijkstra requires nonnegative weights; Bellman-Ford tolerates negatives and detects reachable negative cycles. MST cut and cycle properties justify Kruskal and Prim. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of shortest paths and minimum spanning trees. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative

derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach shortest paths and minimum spanning trees on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from graph traversal and dag algorithms to network flow and reductions. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Relaxation improves a tentative path using one edge.
- **Mechanism.** Dijkstra requires nonnegative weights; Bellman-Ford tolerates negatives and detects reachable negative cycles.
- **Boundary.** MST cut and cycle properties justify Kruskal and Prim.
- **Decision test.** I should be able to say when shortest paths and minimum spanning trees is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I check assumptions before naming an algorithm; one negative edge invalidates Dijkstra's settled-distance proof.

The easy version

Relaxation improves a tentative path using one edge.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why is the lightest edge across an MST cut safe? The conclusion is For any MST missing it, add the edge to create a cycle. That cycle crosses the cut again at an edge no lighter; exchange that edge out without increasing total weight.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why is the lightest edge across an MST cut safe?
Solution. For any MST missing it, add the edge to create a cycle. That cycle crosses the cut again at an edge no lighter; exchange that edge out without increasing total weight.

Practice 2. Give a short oral explanation of shortest paths and minimum spanning trees that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct shortest paths and minimum spanning trees from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a shortest paths and minimum spanning trees problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 7 Network flow and reductions

The idea

A residual graph records unused forward capacity and cancelable previous flow. Augmenting paths increase value; max-flow min-cut certifies optimality. Reductions translate feasible solutions while preserving the answer.

Rebuild the chapter from first principles

The claim I need to own is this: A residual graph records unused forward capacity and cancelable previous flow. Augmenting paths increase value; max-flow min-cut certifies optimality. Reductions translate feasible solutions while preserving the answer. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of network flow and reductions. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach network flow and reductions on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from shortest paths and minimum spanning trees to np-completeness and the limits of efficient search. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** A residual graph records unused forward capacity and cancelable previous flow.
- **Mechanism.** Augmenting paths increase value; max-flow min-cut certifies optimality.
- **Boundary.** Reductions translate feasible solutions while preserving the answer.
- **Decision test.** I should be able to say when network flow and reductions is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

My shorthand for this chapter is: my shorthand for this chapter is: residual backward edges are permission to revise an earlier decision, not flow moving physically backward.

The easy version

A residual graph records unused forward capacity and cancelable previous flow.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Turn bipartite matching into max flow. The conclusion is Connect source to left vertices, bipartite edges left to right, and right vertices to sink, all capacity one. Integral max flow selects vertex-disjoint matching edges.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Turn bipartite matching into max flow.
Solution. Connect source to left vertices, bipartite edges left to right, and right vertices to sink, all capacity one. Integral max flow selects vertex-disjoint matching edges.

Practice 2. Give a short oral explanation of network flow and reductions that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct network flow and reductions from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a network flow and reductions problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 8 NP-completeness and the limits of efficient search

The idea

P contains polynomial-time solvable decision problems; NP contains problems with polynomially verifiable certificates. To prove NP-completeness, show membership in NP and reduce a known NP-complete problem to the new problem in the correct direction.

Rebuild the chapter from first principles

The claim I need to own is this: P contains polynomial-time solvable decision problems; NP contains problems with polynomially verifiable certificates. To prove NP-completeness, show membership in NP and reduce a known NP-complete problem to the new problem in the correct direction. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state

what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of np-completeness and the limits of efficient search. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach np-completeness and the limits of efficient search on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from network flow and reductions to the cumulative course model. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** P contains polynomial-time solvable decision problems; NP contains problems with polynomially verifiable certificates.
- **Mechanism.** To prove NP-completeness, show membership in NP and reduce a known NP-complete problem to the new problem in the correct direction.
- **Boundary.** The useful test is whether I can apply np-completeness and the limits of efficient search to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when np-completeness and the limits of efficient search is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I say 'if I could solve the target quickly, I could solve the known hard source quickly' to check reduction direction.

The easy version

P contains polynomial-time solvable decision problems; NP contains problems with polynomially verifiable certificates.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Outline a proof that Hamiltonian cycle is in NP. The conclusion is The certificate is an ordering of vertices. Verify every vertex appears exactly once and each consecutive pair, including last to first, is an edge; this takes polynomial time.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Outline a proof that Hamiltonian cycle is in NP.
Solution. The certificate is an ordering of vertices. Verify every vertex appears exactly once and each consecutive pair, including last to first, is an edge; this takes polynomial time.

Practice 2. Give a short oral explanation of np-completeness and the limits of efficient search that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct np-completeness and the limits of efficient search from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a np-completeness and the limits of efficient search problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

Final study check

I should be able to close this packet and reconstruct the core models, not merely recognize their names. My final check is to explain one complete problem aloud: what the inputs mean, which invariant or contract controls the work, why the method is legal, what it costs, how it can fail, and what evidence would convince another engineer.