CS 231: Algorithmic Problem Solving

Spring 2022 Notes By Richard Dong

Module 1: Introduction

Defining Problem

Problem: a way of associating an input and an output. It will include:

- **input specification**: explaining the types of data used
- **output specification**: explaining how the input data is related to what is produced

The problem should be as general as possible (by ignoring unimportant details) and make senses(for any possible input data, there will exist output data that satisfies the output specification).

In addition:

Instance: an instance of a problem is specific data that satisfies the input specification

Solution: A solution to an instance of a problem satisfies the output specification

Specifying a Problem

This is generally done in 4 steps:

- Make the problem general.
- Form the input specification using the most important data.
- Form the output specification using one or more easily measurable goals.
- Make sure that there is output specified for any data that satisfies the input specification.

The algorithm **consumes** the instance and **produces** the output.

Questions to consider when forming a problem

- Can the input set or sequence be empty?
  - If so, add **"False"** as a possible output if needed.
- Can there be more than one solution?
  - If so, use "a" instead of "the".
- Can ordered data items appear more than once?
  - If so, use **"nondecreasing"** instead of "increasing" and **"nonincreasing"** instead of

"decreasing".

Types of Data: Grids

Grid is a way of organizing data items in a rectangular arrangement In a grid:

- there ares **rows** and **columns**
- both start at 0 and end at number of rows/columns -1

Use the python module grids.py.

Types of Data: Trees

For data that is organized as a hierarchy, progress over time, or subdivision of data into groups and subgroups.

For reference: check trees.py

A tree consists of zero or more **nodes**

- in a nonempty rooted tree, one node is designated as the **root**.
- Except for the root, each node has a **parent**, to which it is connected by an **edge**.
- Each edge connects a parent and a **child**.
- Nodes with the same parent are **siblings**.
- In an ordered tree, the children of each node are assigned an **order**.
- In an unordered tree, no order is provided.
  - The relative order of the subtrees of a node is not important
- We can categorize nodes by whether or not they have children:
  - a node without children is a **leaf** and a node with at least one child is an **internal node**.

A **path** in a tree is a sequence of nodes (without repeats) such that there is an edge between each consecutive pair of nodes in the sequence.

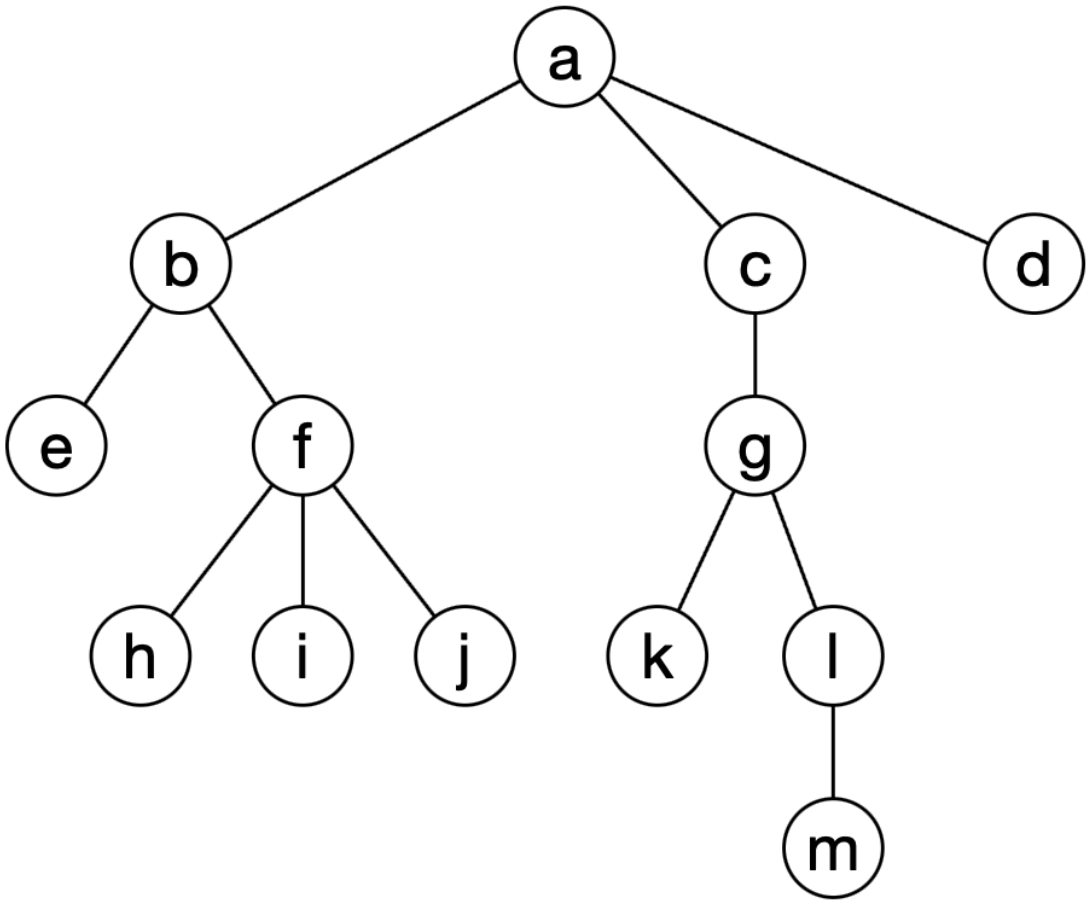
- Equivalently, a path can be defined as a sequence of edges.
- The **length of the path** is one less than the number of nodes
  - that is, the number of edges included in the path
- The first and last node in the sequence are the **endpoints** of the path, and all other nodes are **intermediate nodes** of the path.

To be able to use trees for a wide variety of applications, we may add extra information, such as **weights** or colours, to nodes or edges.

EXAMPLE

In the image below, the root has the label a; the leaves have the labels d, e, h, i, j, k, and m; the children of g have labels k and l; and the nodes with labels b, c, and d are siblings.

The nodes with labels b and m are the endpoints of a path of length 5, since the path is composed of five edges.



Unless stated otherwise, you can assume that each tree has a root and is an unordered tree.

Types of Data: Graphs

For reference: check graph.py

Instead of nodes we have **vertices** (which is the plural of vertex);

we still use the term **edges** for connections between pairs of vertices. As we did for trees, we may add extra information, such as weights or colours, to vertices or edges.

Here, the only important information is which pairs of vertices are connected by edges.

We also have **paths** in graphs. The **endpoints** are now vertices, and vertices that are not endpoints are called **intermediate vertices**. The length of a path is one less than the number of vertices, or the number of edges.

Unlike a tree, we don't have a guaranteed number of edges: there can be no edges at all, every possible edge, or anything in between.

- **complete graph**: a graph in which there is an edge between any pair of vertices.
- **connected graph**: a graph in which there exists a path from any vertex to any other vertex.
- **simple graph**: a graph without vertices connecting to themselves, or more than one path connecting between vertices
- **undirected graph**: There is no order or direction between vertices

The endpoints of a path might be connected by an edge outside of the path. If that is the case, the vertices form a **cycle**.

Vertices that are endpoints of an edge are called **adjacent**. **Neighbour** of an vertex are all adjacent vertices. The set of neighbour is called **neighbourhood**.

We can characterize the vertex by the number of neighbours that it has, or **degree**.

Incidents are used to describe two edges that share an endpoint, and also the relationship between an edge and a vertex that is one of its endpoints.

Terminologies for describing a graph

- V(G): set of vertices of G
  - Size of V(G) is n
- E(G): set of edges of G
  - Size of E(g) is m
- ab or {a,b}: an edge between a and b

Unless stated otherwise, you can assume that each graph is simple and undirected.

Types of Data for Input and Output Specification

Input Specification

- Numbers
- Strings
- Sets
- Sequences
- Grids
- Trees
- Graphs

Output Specification

- Ordering of data items
- Categorization of data items
- Subset of data items

Instance Size

Formally, the size of the instance is the number of bits used in the **encoding**. The number of bits depends on the number of data items and their types.

We will use more imprecise measures of instance size to classify runtimes:

- Number: Where the instance is a number that can be very large, the variable n is the **size** of the number.
  - In contrast, when a number is a bound that may be dominated by the size of other data in the instance, it is considered to be of constant size.
- String: The variable n is the **length** of the string.
- Set: The variable n is the **number of elements** of the set.
- Sequence (string, Python list, Python tuple): The variable n is the **length of the sequence**.
- Grid: The variable r is the **number of rows** and the variable c is the **number of columns**.
- Tree: The variable n is the **number of nodes** in the tree.
- Graph: The variable n is **number of vertices** and the variable m is **number of edges**.
- Multiple types of data: If the instance consists of a multiple types of data, **a constant size data item can be ignored**.
  - For example, if the instance is a graph and an integer bound, we use n and m for the graph and do not count the size of the integer separately.

Optimization Problems

Optimization Problem: Any problem that attempts to find an entity that is the best in some way

- It is a **minimization problem** if the goal is to find a feasible solution with minimum value
- It is a **maximization problem** if the goal is to find a feasible solution with maximum value.

Feasible Solution: the type of information that we're trying to produce. Some feasible solution might be better than the others; informally also known as **entity**.

- **Optimal solution**: a feasible solution that has a numerical value that reaches our goal

Recipe for defining an optimization problem

1. Define a feasible solution.
2. Provide a way of calculating a numerical value for each feasible solution.
3. Specify whether the goal is maximization or minimization.

Types of Optimization Problems

constructive optimization problem: an optimal (the best) solution.

evaluation optimization problem: the value of an optimal solution.

Always make sure that there is an output specified for any data that satisfies the input specification

Decision Problems

The output of a decision problem is the answer to a yes/no question

- **yes-instance**: the output is yes
- **no-instance**: the output is no

To convert an optimization problem into a decision problem, we typically add a bound that we can use in the question.

- Input specification: your original input specification and a number n
  - For minimization: is there a value at most of a number n?
  - For maximization: is there a value at least of a number n?

may be able to stop as soon as a **witness** has been found

- witness: information indicating which solution is correct

Search Problems

Output is an entity that satisfies the specified conditions

- a constructive optimization problem can be viewed as a special type of search problem
  - In this course we will reserve the term search problem for problems that are not constructive optimization problems
- One natural way to form a search problem is from a decision problem

Counting and Enumeration Problems

The output of a **counting problem** is the **number of entities** that satisfy the specified conditions.

The output of an **enumeration problem** is **all the entities** that satisfy the specified

conditions.

- The set of all entities that satisfy the conditions

Paradigms

Basic techniques or approaches that can be used for many problems.

- Make sure that you are using the paradigm specified, even if it results in a cumbersome algorithm.
- Not all paradigms work for all problems.
- A paradigm can be used to create multiple, different algorithms for a problem.
- An algorithm developed using a particular paradigm may not be fast.
- An algorithm developed using a particular paradigm may not produce the correct output.
- Important aspects to consider
  - designing algorithms,
  - analyzing an algorithm for **correctness**,
  - analyzing an algorithm for **running time**, and
  - implementing an algorithm.

Paradigms for Exhaustive Search

Solving a problem by searching through all the possibilities.

Sketch

1. Generate all possibilities
2. Extract information from one possibility at a time
3. Determine the solution from the extracted information

Checklist

- Definition of set of possibilities
- Process for generating all possibilities or the next possibility
- Definition of information to extract
- Process for extracting information
- Process for forming the solution from all or some of the extracted information

Module 2: Order Notation

Functions

We are using functions to express **running times**, where typically the variable n is used to

express the **size of an instance**.

- ex: f(n) = n/2 means that the running time is half of the instance

We care about how does the value of the function depend on the size of n.

Floors and Ceilings

- **Floor**: produce the nearest integer below the symbol
- **Ceilings**: produce the nearest integer above the symbol

When n is even: floor of n/2 = ceiling of n/2 = n/2 When n is odd: floor of n/2 = n/2 - 1/2; ceiling of n/2 = n/2 + 1/2

floor of n/2 + ceiling of n/2 = n (regardless of whether n is odd or even)

Exponents and Logarithms

In computer science, **log means log2**.

- For log n --> Think about how many times you can divide n by 2 to reach 1
- If we are taking the ceiling of log(10) --> we round up to the next cloest interger
- Some Log Laws:
  - loga x = logb x / logb a
  - log xy = log x + log y
  - log (x/y) = log x - log y
  - log x^k = k log x

Sets

A **set** contains at most one copy of an item

- We use $x \in S$ to indicate that the item x is contained in the set S
- **Size of set**: the number of items, or elements, in the set
- **empty set**: has size zero (denote as {})
- **subset**: The set S is a subset of the set T, written as $S \subseteq T$, if every element of S is also an element of T
- If $S \subseteq T$ and $T \subseteq S$, then S = T
- If there exists an element of S that is not in T or an element of T that is not in S, then $S \neq T$
- $S \subset T$ indicates that every element of S is contained in T but that S and T are not equal
  - T contains an element that is not in S
- **Union ($\cup$)**: the union of S and T consists of every element that is in **at least one** of S and T

- **Intersection (∩)**: The intersection of S and T consists of every element that is in both S and T

Sum, Product, Maximum, Minimum, Bounding Quantities, and Ordering

∑ is a Greek equivalent of S for sum, and that Π is a Greek equivalent of P for product.

We use min{set} and max{set} to find the max and min of a set

- Or, minS{i}, the min from the set S

An **arithmetic sequence** is a sequence of numbers in which each number differs from the previous number by the **addition** of some fixed quantity

- Sum from i=1 to i=n is n(n+1)/2

A **geometric sequence** is a sequence where each number in the sequence differs from the previous number by the **multiplication** of some fixed quantity.

- sum from i=1 to i=n = ((1-c^n)/(1-c))

Bounding Quantities

- To show x is at most z --> showing that $x \leq y$ and that $y \leq z$
- If you already know that $x \leq y$, you can also conclude that $x \leq ya$ for $a \geq 1$ and that $x \leq y+b$ for $b \geq 0$.
- you already know that $x \leq y$, you can also conclude that $x/c \leq y$ for $c \geq 1$ and $x-d \leq y$ for $d \geq 0$.

Permutation is an ordering of elements

- the number of possible assignments of elements to categories
- n **factorial** and defined as $n!$

A **combination** is a way of selecting items from a collection, where the order does not matter.

Running Time as a Function

Express the running time of an algorithm as a **function** f(n), where n is the instance size (or, equivalently, the input size)

- the smaller function is a faster algorithm
- Not that larger instance size usually gives longer running time --> so make sure you are taking about size of instances as a function --> DO NOT answer: for which isntances

does the algorithm run fastest (as the number is 1)
    - That is, do no attempt to fix instance size!

Probability:

- When we have a set of possibilities (more formally, events), we can express how likely each event is by assigning it a probability
- The assignment of a probability to each event is a **probability distribution**
- When the probabilities are all equal, as in this case, the probability distribution is known as a **uniform distribution**.
- **Average case**: The value of f(k) is the sum over all instances I of size k of the **probability of I** multiplied by the **running time** of the algorithm on instance I.

Best case running time: When we choose the **smallest** value as the representative for each instance

Worst case running time: If we choose the **largest** value as the representative for each instance

- we usually use worst case running time

We can often show upper or lower bounds without constructing actual instances.

Each type of running time is determined by choosing a **representative** value for each instance size.

- For any instance size, the representative value:
    - best case <= average case <= worst case

Categorizing Functions

Category of Functions: Depend on how n is operated on

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential

Dominant: a term that is biggest as a function of n

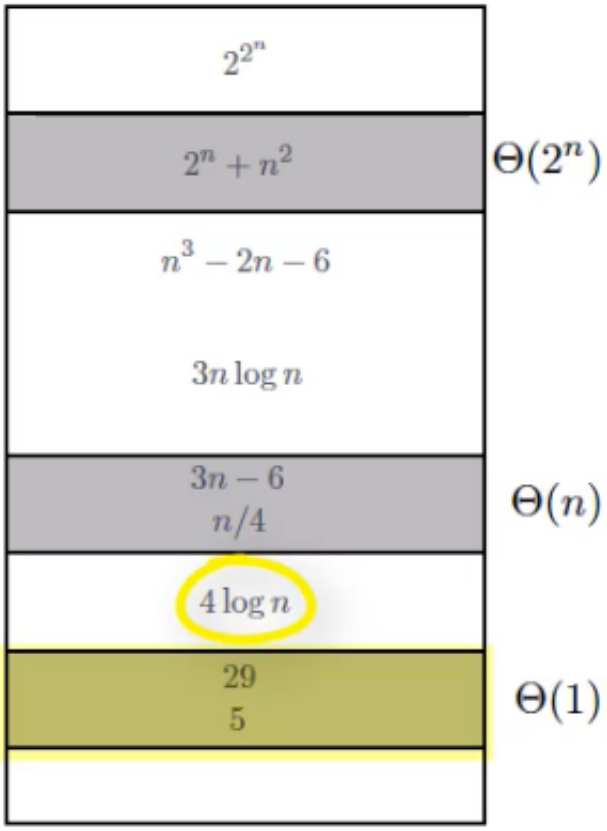
Simple Function: no multiplicative factors or additive terms

- Constant: 1
- Logarithmic: 1og(n)
- Linear: n

- Quadratic: n^2
- Exponential: 2^n

We use $\Theta(g(n))$ to define a set in terms of the **simple function $g(n)$**.

- Theta represents both the upper bound and the lower bound
- We can then have members of a catagory, where $f(n) \in \Theta(g(n))$
- They are in the catagory because g(n) is the dominant term



Multiple Variable Functions:

- the size of a **grid** is defined in terms of both the number of rows and the number of columns
- the size of a graph is defined in terms of both the number of vertices and the number of edges
- Simple Functions on Two Variables (m and n)
  - 1, log n, n, n^2, log m, m, m^2, m+n, m * n, n log m, $m^{2}n$^3
- To make sure that we represent the function property, we retain any term that might be dominant.
  - you may not know how m and n is related, so only eliminate the dominant term of the same variable

Catagorizing functions with partial information

- We can think of the line at the top of the rectangle for $\Theta(g(n))$ as indicating an upper bound based on $g(n)$ and the line at the bottom of the rectangle for $\Theta(g(n))$ as indicating a lower bound based on $g(n)$
- Refering to the run-time box diagram above:
  - $f(n)$ is in $\boldsymbol{O(g(n))}$ (pronounced "Big Oh" of $g(n)$) if $f(n)$ is **below** the line at the top of the rectangle for $\Theta(g(n))$ --> upper bound
  - $f(n)$ is in $\boldsymbol{\Omega(g(n))}$ (pronounced "Big Omega" of $g(n)$) if $f(n)$ is **above** the line at the bottom of the rectangle for $\Theta(g(n))$ --> lower bound
  - if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then $f(n) \in \Theta(g(n))$
  - if $f(n) \in \Theta(g(n))$, then $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

If g(n) < h(n):

- If $f(n) \in O(g(n))$, then $f(n) \in O(h(n))$.
- If $f(n) \in \Omega(h(n))$, then $f(n) \in \Omega(g(n))$.

All O, theta, and omega uses simple functions

Order Notation Formalism

We do not realy care about small n, we care more about the long term behaviour of f(n). In addition, we only care about the shape, rather than the specific values

Formal Definitions of Asymptotic (Order) Notation:

- $f(n)$ is in $\boldsymbol{O(g(n))}$ if there is a real constant $c>0$ and an integer constant $n0 \geq 1$ such that $\boldsymbol{f(n) \leq cg(n)}$ for every $n \geq n0$.
- $f(n)$ is in $\boldsymbol{\Omega(g(n))}$ if there is a real constant $c>0$ and an integer constant $n0 \geq 1$ such that $\boldsymbol{f(n) \geq cg(n)}$ for every $n \geq n0$
- $f(n)$ is in $\boldsymbol{\Theta(g(n))}$ if there are real constants $\boldsymbol{c1>0}$ **and** $\boldsymbol{c2>0}$ and an integer constant $\boldsymbol{n0 \geq 1}$ such that $\boldsymbol{c1g(n) \leq f(n) \leq c2g(n)}$ for every $n \geq n0$.

Some Important Ideas:

- The constant $n0$ is used to ignore how the function behaves on small values.
- The constants c , $c1$, and $c2$ can be less than 1.
- If you have constants to show that $f(n) \in O(g(n))$ and that $f(n) \in \Omega(g(n))$, the **larger of the $n0$ values** will hold for both.
- For a particular pair of functions $f(n)$ and $g(n)$, there may be many possible choices of constants that fit the definition.
- When proving using the formal definition, make sure that explicit constants c and n0 are used

Analyzing Running Times

For Sums of Functions: keep only the dominant term (or terms, if there are multiple variables) among the categories

- use Θ if all the chosen categories (dominant ones) use Θ, and
- use O if any one of the chosen categories (dominant ones) uses O

For Products of Functions: take the product of the categories, keeping only the dominant term or terms if there are additions

- use Θ if all the categories use Θ , and
- use O if any one of the categories uses O

For Multiple Variables:

Suppose we have two variables, m and n .

- If we can express m as $\Theta(g(n))$, then we can substitute $g(n)$ for m anywhere in the simple function, and then simplify the simple function to remove any terms that are dominated.
- If we can express m as $O(g(n))$, the upper bound on m might allow us to remove terms using m as no longer dominant.
- If we can express m as $\Omega(g(n))$, the lower bound on m might allow us to remove terms that are dominated by the terms using m .

You can use any kind of order notation (O, Ω, or Θ) to classify any kind of running time (worst-case, best-case, or average-case running time).

Variables must remain variables.

A variable n does not need to appear in the expression of a function on n ; for example, if the function does not depend on n at all, a constant function can be used.

Runtime for permutation:

- The function n! is greater than 2n and smaller than n^n .
- nPk $\in \Theta(nk)$

Module 3: Pseudocode

Describing Algorithms

Goal: be able to describe algorithms in enough detail that we can **compare their running times** without investing the time to code them

- Describe an algorithm in a way that **does not** depend on the choices of programming language, operating system, or hardware.
- Use the algorithm description as a way to get a rough estimate of the running time that

can be **compared** to other algorithm ideas.

Models of Computation

a set of operations and the resources they require, such as time and space.

- In this course, we will focus on **time requirements**.
  - the time requirements will not typically equal those of an actual program due to the choice of different hardware, operating systems, and programming languages
  - obtain rough upper and lower bounds, using order notation

Random Access Machine (RAM)

- A commonly-used model of computation
- consists of a set of **registers**, used to store values, and a set of possible instructions
- A RAM algorithm consists of a **sequence of instructions**, which can be combined to form familiar program structures such as branching and looping.

Pseudocode

In analyzing pseudocode, we are implicitly making assumptions about **costs of operations on a RAM**, where simple operations can each be executed in constant time

- assignment of a value to a variable
- use of a variable
- moving to another point in the program
- simple arithmetic and Boolean operations.

The cost of pseudocode approximates the cost of an actual program.

For this course, pseudocode contains:

- An **algorithm sketch**, using sentences or point form to get across the basic ideas (usually from the suggested paradigm)
- Descriptions in high-level pseudocode, to show the structure of the algorithm without going into details
- Descriptions in detailed pseudocode, to allow line-by-line analysis
  - limit yourself to the allowed operations, methods, and functions, as detailed on the reference page on costs.

For style of pseudocode, see Pseudocode used in this course.

Analyzing Runtimes

Single Line

- need to know the cost of each operation, method, or function used in the line
  - assume that each operation, method, and function requires time at least Θ(1)
- If we repeat constant-time steps a non-constant number of times, such as in a loop, then their cost might dominate the cost of other steps.
- Do not assume that Python list operations, methods, and functions can be executed in constant time. Examples such as sort, map, filter, and reduce require time that depends on the length of the list.
- The running time of many Python dictionary operations, functions, and methods also depend on the **size** of the dictionary.
- If an operation, method or function does not appear on the reference page on costs, it probably should not be used in an assignment question.
- Some constant time steps:
  - Assigning a value to a variable
  - Using a variable
  - Using an arithmetic or Boolean operation or a comparison
  - Moving to another line in the program
  - Returning a value using return
- **The function by itself does not incur any cost, only when it is called/run**

Block of Code

- If each block is executed once, the total cost will be the sum of the costs of the blocks.
- Since we are using order notation, we will be retaining only the dominant term or terms.
- Branching (if-statements) in blocks:
  - consider that some lines of code may not be executed and the costs of the conditions that are being checked.
  - the total cost is determined by adding the costs of the lines in the branch that is executed, including the costs of any conditions that are evaluated to reach that branch

Loops

- add up the costs of all iterations
  - the cost of an iteration as being the sum of cost of **iteration management** (the cost of ensuring that the loop is executed the correct number of times) and the cost of the **loop body** (the cost of the indented lines executed during each iteration)
- Common situations: make sure you explain which it is and why before you apply the cost
  - The cost of each iteration is asymptotically the same for each iteration (that is, each iteration has a cost in Θ($g(n)$) for some function $g(n)$).
    - For this situation, the cost of the loop is the **product of the number of iterations and the *cost* of one iteration**.
  - The cost of iteration i is in Θ(i) .
    - For this situation, the cost of the loop is **Θ(k^2)** , where k the number of

iterations

- In general:
  - For worst-case analysis
    - one option is to obtain an upper bound using O by multiplying **the number of iterations** by the cost of the **most expensive iteration**, where the cost of each iteration is the sum of the cost of iteration management and the cost of the loop body.
  - For best-case analysis
    - find an instance on which the number of iterations is less than the maximum possible, and for which the cost of a loop is less than the maximum possible.
    - What makes the analysis tricky is that in order to base the analysis on a smaller number of loops and a smaller cost for some iteration, one needs to be sure that **there exists an instance for which both occur**.

Complete Functions

- challenging, since not every block of code will be executed for every instance
  - We cannot in general simply sum up the costs of a sequence of blocks
- If we can express running time using Θ instead of O , we will do so. If this is not possible, you should use O, but with the smallest function possible
- Recipe for analyzing worest-case running time
  - Break each block into blocks.
  - Determine a bound on each block individually.
  - Retain all dominant costs.
  - Use Θ if all costs are expressed in Θ and can occur simultaneously and use O otherwise.
- If you introduce **temporary variables** in the analysis of an algorithm, be sure to remove all temporary variables from the final result.

Module 4: Greedy Algorithm

Motivation

While exhuastive search is great, it is very costly since exhaustive search requires generating all possible feasible solutions, and the number of feasible solutions is large.

Paradigm for Greedy Algorithms

A greedy algorithm builds up a solution **step by step**, often using an ordering to make a decision.

- The term "greedy" captures the idea of building up the solution by grabbing **whatever seems to be the best at the moment**.

Preprocessing Stage:

- Organizing all the data at once

Sketch for Greedy Algorithm

- Process data
- Loop to build up the solution step by step
  - Use information to make a decision
  - Make updates to information

Checklist for Greedy Algorithm

- Process for preprocessing data
- Definition of steps needed to build a solution
- Definition of information to be used for a decision
- Definition of criteria for a decision
  - In this course, we'll use the convention that unless specified otherwise, ties will be broken **arbitrarily**
    - Note that his is not **random**: we still have rules, they just can be any rules. The term random means that you are using randomization to make a choice.
- Process for making updates to information

The greedy algorithm isn't guaranteed to produce the correct solution, even though it does run faster.

Logics and Proofs

Statements:

- statement can be either **true** or **false**
- Logic operators:
  - A OR B, which is true when A is true, B is true, or both A and B are true and false otherwise
  - A AND B, which is true when both A and B are true and false otherwise
  - NOT A, which is true when A is false and false otherwise

Predicates

- an expression that refers to at least one variable
  - ex: P(x) = "The number x is odd."

Existential Statements

- the predicate is true for at least one value in the set

- "There exists an x in the set S such that $P(x)$ is true."

Universal Statements

- the predicate is true for all values in the set
- "For every x in the set S , $P(x)$ is true."

To prove that **an algorithm is correct**, we will typically prove a **universal statement**, where S can be viewed as the set of instances and $P(x)$ can be viewed as the statement that the algorithm performs correctly on instance x.

In contrast, to prove that **an algorithm is not correct**, we will typically prove an **existential statement**, where S is again the set of instances but now $P(x)$ is the statement that the algorithm fails to perform correctly on instance x .

Negation

- When A is a simple statement, we can usually form the negation by adding NOT or by replacing true with false.
- To negate an **existential statement**, we state that there **does not exist** any value in the set for which the predicate is true.
  - "For all x in the set S , $P(x)$ is false."
  - We have just transformed an existential statement about $P(x)$ into a universal statement about the negation of $P(x)$.
- To negate a **universal statement**, we state that the predicate is **not true for every value** in the set. That is, we state that there exists a value in the set such that the predicate is not true.
  - "There exists an x in the set S such that $P(x)$ is false"
  - We have just transformed a universal statement about $P(x)$ into an existential statement about the negation of $P(x)$.
- Negation not only makes existential statements into universal statements and vice versa, but also **exchanges ORs and ANDs**.
- The negation of a true statement will be a false statement, and the negation of a false statement will be a true statement.

Implication

- the statement " A implies B " means that if A is true, then B is true.
- The implication says nothing about what happens when A is false, so if A is false, B can be either true or false
- The **negation** of " A implies B " is " A is true and B is false."

Converses

- We can form the converse of an implication " A implies B " by **swapping A and B** to

form " B implies A "
- an implication may be true without its converse being true

Contrapositives

- the contrapositive of " A implies B " is the implication "not B implies not A "
- An implication and its contrapositive are always equivalent (they are either both true or both false)

If " A implies B " and " B implies A ."

- both an implication and its converse are true
- " A if and only if B "
  - " A if B " is another way of saying " B implies A " and
  - " A only if B " is another way of saying " A implies B ."
- prove " A if and only if B " by proving both " A implies B " and "not A implies not B ."

Proofs

- a statement is true whenever its negation is false, therefore:
  - you can prove that a statement is true by showing that its negation is false, or
  - you can prove that a statement is false by showing that its negation is true.
- **modus ponens**
  - if A is true and " A implies B " is true, then B is true

Proof by Contradiction

- A statement cannot be both true and false. If we can show that a statement is both true and false, we have reached a contradiction.
- To show X is true:
  - Suppose that X (which we wish to prove) is false.
  - Choose a statement Y that we know to be true.
  - Use the assumption that X is false to show that Y is false.
  - Since Y being both true and false is a contradiction, we know that X is true.

Proving a Univerisal Statement

- we need to show that $P(x)$ is true for every x in S .
- choose a generic x in S and show that $P(x)$ is true.

Proving an Existential Statement

- Choose an x .
- Show that x is in S .
- Show that $P(x)$ is true.

Proving an algorithm is not correct

To prove that an algorithm is **correct**, we need to prove a statement of the form "On **any instance**, the algorithm produces the correct output."

To prove that an algorithm is not correct, we need to prove **its negation**: "There exists **an instance** on which the algorithm produces an output that is not correct."

- Choose an instance x .
  - The instance x is called a **counterexample**.
  - Maybe asked to show:
    - an example that fails on at least one sequence of arbitrary choices when breaking ties, or
    - an example that fails on every sequence of arbitrary choices when breaking ties.
- Show that x is an instance of the problem the algorithm is supposed to solve.
- Show that the algorithm produces y .
- Show that the correct solution to the problem on x is z , where z is better than y .

Examples of Greedy Algorithm

- **Kruskal's algorithm**
  - relies on a way to group vertices into *sets*, where the vertices in a set form a tree. Checking to make sure that no cycle has been formed is as simple as checking to ensure that the endpoints of each added edge are in different sets.
  - worst-case running times: $O(m\log m)$
    - n is the number of vertices and m is the number of edges.
- **Prim's algorithm**
  - relies on a way to update the cheapest edge connecting a vertex *outside the tree* to a vertex in the tree.
  - worst-case running times: $O(m\log n)$
    - n is the number of vertices and m is the number of edges.

Proving the correctness of a greedy algorithm

Two properties must be satisfied:

- **Optimal Substructure Property**:
  - A problem satisfies this property when the optimal solution to an instance of a problem can be formed from optimal solutions to one or more smaller instances formed from the original instance.
- **Greedy Choice Property**:
  - A greedy algorithm satisfies this property when there is an optimal solution

consistent with each greedy choice made by the algorithm.

Summaries

- Properties:
  - Each step of the algorithm eliminates some possible solutions from consideration, as no decision is ever "undone"
  - Not all problems can be solved using greedy algorithms.
  - Proving correctness can be difficult.
  - Proving an algorithm is not correct can be achieved with a single **counterexample**.
  - Running time analysis is often simple.

Module 5: Divide and Conquer

Paradigm

The term **divide** comes from the fact that we divide an instance of the problem into two smaller instances and the term **conquer** comes from the fact that we then go on to solve, or conquer, the smaller problems. Then we **combine** the solution or solutions of the smaller instance or instances to form a solution to the original instance.

Recursion

a recursive definition consists of at least one **recursive case** and at least one **base case**.

- A recursive case defines an entity in terms of one or more entities of the same type, each of which are closer to the base case.
- The base case, in contrast, is defined without any reference to entities of the same type.

a recursive function consists of one or more base cases, solved directly, and one or more recursive (or general) cases, solved using the function itself.

Important Caution

- A function call to an input the same size (or the input itself) leads to an infinite loop.
- The absence of a base case may mean that the computation never ends.
- Base cases and recursive cases together must cover all possible sizes of inputs.

Algorithm Sketch

- Solve base cases directly
- Divide into smaller instances
- Conquer recursive cases using recursive calls
- Combine results on smaller instances

Checklist

- Definition of smaller instances
- Definition of base cases
- Process for dividing the instance
- Process for combining results

an easy divide step resulted in a not-so-easy combine step

Recurrence Relations

A **recurrence relation** (or recurrence for short) is used to express the running time of an algorithm on an input of size n , written as $T(n)$, for all values of n . This is typically expressed as:

- For small values of n , base cases expressed as a function of n .
- For larger values of n , a recursive case expressed as a **function of n** and of $T(k)$ for one or more values of k smaller than n .
  - For the recursive case, the cost can be viewed as the sum of the divide step, the conquer step, and the combine step.
  - The cost of the **conquer** step will be written using $T(k)$ for one or more values of k smaller than n .
    - k might be expressed using floor or ceiling notations

To be able to compare algorithms with running times expressed as recurrence relations, we'd like to be able to determine the category using order notation. To do so is know as putting the recurrence in **closed form** or, equivalently, **solving the recurrence**. (ie: without the use of T on the right hand side)

For greater efficiency of both space and time, we can instead consider making **in place** computations, where we **access and modify** a subsection of a list, such as by specifying the starting and ending indices of the subsection of interest.

- We can then avoid the linear cost of creating a slice.

Three major methods for solve recurrances are

- Iteration method
- Master theorem method
- Substitution method

Iteration Method

- Apply the definition of $T(n)$ **iteratively**, expressing $T(n)$ in a general form for any number of iterations.

    - whatever is doing to n to reach the base case, keep doing it
- Choose a number of iterations that reduces any smaller term to the base case.
- Express $T(n)$ in closed form.

trying to manipulate order notation directly in the iteration method can be dangerous, and should be avoided.

- If, in an assessment, you are asked to use the iteration method to prove a closed form is in $O(g(n))$, use a specific function $f(n) \in O(g(n))$ to prove the result

Master Method

It does not work for all recurrence relations

- It only works on the folloing form:

$$T(n) = aT(n/b) + f(n)$$

  where a>= 1 and is a constant, b>=1 and is a constant, and we ignore floors and ceilings
- we treat both the floor and celing as if they were not floor or ceiling
- $f(n)$ represent the divide and combine step

To determine the closed form of a recurrence of the form $T(n) = aT(n/b) + f(n)$, the Master Theorem shows that it suffices to compare the values of $f(n)$ and $x = n^{log_b^a}$

- $f(n)$ is "smaller than" x if $f(n) \in \Theta(g(n))$, $x \in \Theta(h(n))$ and $\boldsymbol{g(n) < h(n)}$, and
- $f(n)$ is "bigger than" x if $f(n) \in \Theta(g(n))$, $x \in \Theta(h(n))$ and $\boldsymbol{g(n) > h(n)}$.

Proof By Induction

To establish the closed form of a recurrence relation, we wish to prove a universal statement of the form "For every integer $n \geq b$, the predicate $P(n)$ is true."

- Typically the predicate $P(n)$ is of the form $T(n) = f(n)$, $T(n) \leq f(n)$, or $T(n) \in O(n)$ for some function $f(n)$.

When proving predicates for all possible instance sizes, one way to handle such a proof is to use **induction**.

- The universal statement "For every x in S , $P(x)$ is true" can be viewed as an infinite sequence of statements $P(0)$, $P(1)$, $P(2)$, and so on. We wish to show that each of these statements is true.
- We will prove the statement in order, starting with $P(0)$. We will use the fact that $P(0)$ is true to show that $P(1)$ is true, use the fact that $P(1)$ is true to show that $P(2)$ is true, and so on.
    - proving that $P(0)$ is true, known as the **base case**

- climbing the ladder from P(i) to P(i+1), known as the **induction step**
  - show that for any i, P(i) implies P(i+1)
- using modus ponens to put the steps together
  - P(i) is true and P(i) implies P(i+1), then P(i+1) is True

Proving the induction step entails making use of a statement known as the **induction hypothesis**, which is equivalent to assuming that P holds for smaller values in order to prove that P holds for larger values.

Formal Statement of the Master Theorem

Suppose $T(n) = aT(n/b) + f(n)$ for constants a>=1 and b>1:

- if f(n) is in $O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then T(n) is in $\Theta(n^{log_b a})$
- if f(n) is in $\Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) <= cf(n)$ for some constant of $c < 1$, then T(n) is in $\Theta(f(n))$
- If f(n) is in $\Theta(n^{log_b a})$, then T(n) is in $\Theta(n^{log_b a} log(n))$

Substitution Method

Typically, the method is used to prove an **upper bound** on a closed form of a recurrence.

- Such a bound can be viewed as a universal statement of the form "For all positive integers n , $T(n) \leq f(n)$."

We start with the **induction hypothesis**, assuming that the statement holds for smaller integers. The name of the method comes from the fact that we substitute $f(k)$ for each $T(k)$, $k<n$, in the recurrence relation and then show that $T(n) \leq f(n)$. We typically **complete the base cases last**, verifying that $T(n) \leq f(n)$ for values of n covered by the base cases.

The first step of the substitution method is to guess a possible upper bound on $T(n)$ and, through the process of trying to prove that it is an upper bound, gradually refine the guess until the proof is successful.

- In this course, you will be provided information about the upper bound on $T(n)$ to use

Recipe

- Guess an upper bound for $T(n)$.
- Substitute the guess for uses of T on smaller values.
- Simplify the right hand side to prove the bound.
- Check that the bound holds for the base cases.

Substitution and Order Notation

To show that $T(n) \in O(g(n))$, **do not** try to use $O(g(n))$ directly in the recipe.

- Instead, choose a function $f(n) \in O(g(n))$ and use the substitution method to show that $T(n) \leq f(n)$.

General ideas:

- Do not use order notation in the guess in Step 2.
- Use placeholders for constants.
- Adjust and repeat as needed.
- Choose specific values for all placeholders.
- Make sure the general case and base cases cover all values.

Summary of Devide-and-Conquer

- An algorithm is created by decomposing an instance into smaller instances of the same problem, ideally in such a way that the cost of merging and combining is not too costly and the smaller instances are of roughly the same size.
- Divide-and-conquer works primarily for problems with instances that can be decomposed (e.g. a set, entries in a sequence or grid).
- Correctness depends on ensuring that the correct smaller instances have been formed.
- Running time is determined using a recurrence.

Module 6: Dynamic Programming

Motivation: Matrix Multiplication

Some Definitions:

- The *matrix order*, namely, $M_0 M_1 M_2 M_3$, is fixed for a given chain of matrices. It is part of the input specification, and is not changed by the algorithm.
- A *parenthesization* is a way of **grouping matrices** using parentheses. There may be more than one parenthesization for a particular matrix order.
- The *multiplication order* is the order in which the multiplications take place. There may be more than one multiplication order for a particular parenthesization.

When multiplying M1 and M2, the column of M1 should requal to the rows in M2

Goal: multiply a sequence of matrices in the cheapest way possible

- A parenthesization of the matrices $M0$, ... forming an order of multiplication that results in the smallest number of multiplications of pairs of values, where matrix M_i has dimensions $di \times di+1$ (rows time columns)
  - cost is the product of the number of rows in the first matrix, the shared dimension, and the number of columns in the second matrix

We want to use a table to store the information we have already known

Dynamic Programming

The ideas of solving smaller instances, storing them in the table, and then looking them up to solve larger instances, is known as the dynamic programming paradigm

Sketch

- Create a table
- Loop over table entries
  - Fill in base cases
  - Fill in non-base cases
- Extract solution

Checklist

- Definition of the solution in terms of solutions to smaller instances
- Definition of information to store in each table entry
- Definition of base cases and their values
- Definition of the shape of the table or tables needed to store the solutions to the smaller instances
- Definition of order of evaluation
- Process for extracting the solution from the table

Analyzing Dynamic Programming

Three Stages:

- Creating the table
- Filling the table
- Extracting the solution

Implementing Tables

Tables should be created as:

- 1D: Grids
- 2D: Grids
- 3D: ThreeD
- Tree: Tree

To store multiple values:

- use multiple tables, one for each type of value, or

- use a single table that stores objects composed of multiple values

Optimal Substructure

The optimal solution to an instance of the problem can be formed from optimal solutions to one or more smaller instances formed from the original instance.

Recipes:

- Using the optimal solution O for an arbitrary instance I, construct one or more smaller instances.
- Decompose O into pieces, one for each smaller instance.
- Show that if any piece O' of O is not an optimal solution for a smaller instance I' of I, then O is not an optimal solution for I.

Choosing Table Shape

- A "triangle" shape may occur when $i<j$ or when $M[i,j]=M[j,i]$.
- We might be able to use a few tables of a smaller dimension if entries depend on only a few rows or a few columns --> erase the previous entries when you do not need them any more
  - The does not apply to the situation in which previous table entries need to be **re-examined** in order to extract the solution.

Order of Evaluation

The key to determining the order of evaluation is to make sure that the entries on which the current entry depends have already been filled.

- If filling depends on j-2 and j-1 and on i and i+1, ensure you fill in all j in increasing order first

Options for 2D table:

- Increasing order by row; within each row in any order by column
- Increasing order by row; within each row in increasing order by column
- Increasing order by column; within each column in any order by row
- Increasing order by column; within each column in increasing order by row

Summary

Properties

- Dynamic programming consists of solving a problem by building up solutions to smaller

instances, not all of which may be used in the solution.
- Problems that satisfy the **Optimal Substructure Property** are most appropriate for this paradigm.
- Dynamic programming works on instances **from small to large**, working "bottom up" instead of "top down".
- Correctness follows from correctly defining how solutions depend on solutions to other instances.
- Running time depends on the size of the table and the cost of filling in a single entry.

Expressing solutions in terms of other solution

- **Set** - Determine an *order* on the elements. Smaller instances are defined using *smaller subsets* of the elements. Bigger instances are formed by adding elements one at a time.
- **Sequence or Grid** - Define a problem in terms of the *position* or positions in the sequence. Smaller instances are at earlier positions. Bigger instances are at later positions.
- **Tree** - Define a problem on a *subtree*. Smaller instances are on smaller subtrees. Bigger instances are on bigger subtrees.

Type of infromation stored

- **Decision problem** - True or False (solutions to smaller instances)
- **Evaluation problem** - Values (solutions to smaller instances)
- **Search or constructive problem** - Information indicating which smaller instances led to the optimal solution

Because the cost of the algorithm will depend on the amount of information stored in each entry (cost of calculation as well as cost of reading time), **often full solutions are not stored**.

A variant on dynamic programming, known as **memoization**, works top down in a manner similar to divide-and-conquer.

Module 7: Hardness of Problems

Complexity

The **complexity** of a problem is a way of describing the worst-case cost of the best algorithm for the problem.

- Knowing the worst-case running time of an algorithm is not sufficient to know the complexity. We need to prove that the algorithm is the best.

Upper and lower bounds

To prove that a problem has complexity $\Theta(f(n))$, we need an upper bound of $O(f(n))$ and a lower bound of $\Omega(f(n))$.

We distinguish between:

- upper and lower bounds on an **algorithm** that solves a problem, and
- upper and lower bounds on a **problem**.
  - An upper bound of $O(f(n))$ on a problem means that **there exists an algorithm** correctly solving the problem that in the worst case runs in time $O(f(n))$.
  - A lower bound of $\Omega(g(n))$ on a problem means that **any algorithm** that correctly solves the problem must use $\Omega(g(n))$ time in the worst case.

Upper bound on problems

An upper bound on the worst-case running time of an algorithm that solves a problem is also an upper bound on the problem.

The algorithm may not be the lowest possible upper bound, as the algorithm may not be the best algorithm.

Lower bound on problems

A lower bound on the worst-case running time of an **algorithm** may **not** be a lower bound on the problem, as it only refers to **one** algorithm. There might exist another algorithm that runs in less time.

Lower bounds are hard to obtain, since they require reasoning not only about all known algorithms, but also all possible algorithms that could exist.

Reasonable running times

Our motivation for determining the complexity of a problem was to know when to stop looking for a better algorithm.

An algorithm that can run in time less than linear in the size of the instance, or **sublinear time**, is especially notable.

Many of our exhaustive search algorithms result in exponential running times. If we can do better, and obtain something **subexponential**, that would be preferable.

What is reasonable

- **polynomial time**: the time that is *at most* polynomial.
- **polynomial size** the size that is *at most* polynomial.
- an algorithm to run in polynomial time if the running time is in $O(n^c)$ for some constant c, when there is a single variable n.

- if there are two variables m and n , the running time is polynomial if it is in $O(m^c n^d)$ for constants c and d

Robustness

One of the main reasons to classify polynomial time as reasonable is because the class of problems with polynomial-time algorithms is **robust**.

Robustness means that a problem doesn't get booted out of the class due to a small change in the definition.

- For example, using a different graph implementation should not result in the complexity of problems changing.

Properties of Polynomials

- The **sum** of two functions that are at most polynomial is a function that is at most polynomial.
- The **product** of two functions that are at most polynomial is a function that is at most polynomial.
- If f and g are both at most polynomial, then so is the application of f to g .

Complexity classes

A set of problems that can be solved **within specified bounds** on resources using a **specific model of computation**.

In this course, the resource will be **worst-case** running time on a RAM. Our main focus will be the categorization of **decision problems** (ie, output is either yes or no).

We can define classes for:

- a type of problem (e.g. decision, counting),
- a model of computation (e.g. RAM), and
- a resource (e.g time, space, size of numbers).

Complexity class P

A decision problem is in P if it can be solved using an algorithm with worst-case running time that is at most polynomial in the size of the input.

- P consists of problems for which there are **polynomial-time algorithms**.
  - an algorithm with worst-case running time that is at most polynomial in the size of the input

A problem is considered to be **tractable** if it is known to be in P and **intractable** otherwise.

- To prove that a problem is tractable, it suffices to demonstrate the **existence of a polynomial-time algorithm**.

Key operations

We can start with the more modest goal of trying to prove a lower bound on all possible algorithms of a particular type that solve the problem.

A **key operation** is a type of step that can be used to represent the other types of steps.

- Often a key operation is itself a **constant-time step**. Counting the number of key operations gives a lower bound on the cost of the entire algorithm, expressed as a function of the instance size.
- Since the larger the lower bound, the more information we obtain, we prefer the linear lower bound obtained by counting comparisons over the constant lower bound obtained by counting the single return statement.

Comparison-based algorithm

The only way that we can extract information from inputs is by comparing one to another.

We can use the term for algorithms that use any of the following type of comparisons: = , ≠ , < , > , ≤ , and ≥.

To prove a lower bound of $f(n)$ on comparison-based algorithms for a problem, we need to show that all comparison-based algorithms must use at least $f(n)$ comparisons in the worst case.

Decision Trees

To be able to count comparisons, we make use the **decision tree**, a model of computation for comparison-based algorithms.

A decision tree is a way of representing the comparisons made in the course of an algorithm. You can think of computation as starting at the root of the tree, and then ending at one of the leaves. Along the way, each internal node corresponds to a comparison (or, equivalently, decision) made along the way.

We can use the number of decisions as a **lower bound** on the cost of reaching a particular leaf. Since we are ignoring all other steps, **we cannot use a decision tree for an upper bound**.

More formally, an algorithm is represented as a tree as follows:

- Each **internal node** in the tree represents a **comparison**.

- Each **leaf** represents an **output**.
- The **children** of a node represent the possible **next courses of action**, depending on the outcome of the comparison.

The worst-case number of comparisons is the length of the longest path from a root to a leaf, which is also the height of the tree. Any input that leads to a leaf at maximum depth is a worst-case input.

Information Theory Lower Bound

If we can find a property that is true for every possible decision tree for a problem, we have a property about every possible comparison-based algorithm that solves the problem.

- Our goal is to prove a **lower bound** on the **height** of any decision tree that solves the problem.

Statement:

- A tree of height h in which each node has at most two children has at most $2h$ leaves.

Therefore,

- If a problem has ℓ possible **outputs**, then any comparison-based algorithm that solves the problem requires time at least $\Omega(\log \ell)$ in the worst case.
  - the logarithm of the number of possible outputs is a lower bound on any algorithm that solves the problem
  - this known as a **decision tree lower bound**, or, interchangeably, as an Information theory lower bound.

Information theory lower bounds apply **only** to comparison-based algorithms. They do not apply if we can compute various functions on the values of inputs.

Searching n values can have n possible outputs, hence $\Omega(\log n)$ is a lower bound.

Sorting n values can have $n!$ possible outputs, hence $\Omega(\log n!)$, or $\Omega(n \log n)$, is a lower bound.

Lower bound techniques

A technique that can be applied to any choice of key operations, including:

- Access one data item (e.g. grid entry)
- Compare two data items (outcome = or ≠)
- Compare two data items (outcome = , < , or >)
- Determine if two vertices in a graph are adjacent

Any correct algorithm relying on the key operation uses at least T steps by providing an

adversary strategy that ensures if an algorithm produces an answer after at most T–1 key operations, it can be forced to be providing the incorrect solution for at least one input.

Adversary Strategy is a procedure that produces an answer to each question by the algorithm, such that:

- the answer is consistent with previously-given answers,
- there is at least one input consistent with all the answers given,
- there are no limits on time to compute an answer, and
- there are no limits on amount of extra information to store;

but

- there is no knowledge of the algorithm or future questions.

To form a lower bound, we need:

- an adversary strategy, and
- a proof that if the algorithm uses at most T–1 key operations, it will lose.

Recipe for determining an adversary lower bound

- Specify an adversary strategy.
- Determine a number of steps T that any correct algorithm must take.
- Show that after T–1 steps of any algorithm, there will be at least two inputs consistent with the answers given by the adversary, and that they yield different outputs.

Reduction

We could then prove that the algorithm is a polynomial-time algorithm by proving each of the following statements:

- Each helper function has worst-case running time that is at most polynomial in the size of the input.
- Each helper function is executed on an instance of size at most polynomial in the size of the input to our problem.
- Each helper function is executed at most a polynomial number of times.
- All other steps of the algorithm can executed in at most polynomial time.

If we can construct an algorithm using at most a polynomial number of uses of polynomial-time helper functions, the algorithm is a polynomial-time algorithm.

Problem A can be **reduced** to problem B if we can solve A using a procedure for B at most a polynomial number of times and at most polynomial extra time.

- The description of how to solve A using B is known as a **reduction**.

- We say that A is **reducible** to B.
- We **do not need** to know any details about the procedure for B. Although we know that it is correct, we do not need to know how it works nor what its running time is.

Easy A, Easy B

If A can be reduced to B and B is "easy", then A is "easy" too.

- When we are able to provide **reductions in both directions**, we say that the problems are **equivalent**.
- But, it is possible that B is hard yet A is easy
  - You cannot conclude that if A is "easy" then B is "easy".
    - There might be a polynomial-time algorithm for A that does not use B.

Easy B, Easy A

If A can be reduced to B and B is "easy", then A is "easy" too.

Verification Algorithm

A **certificate** for a yes-instance is information that can be used to verify that the yes-instance really is a yes-instance.

Formal definition of verification

A polynomial-time verification algorithm for a decision problem is a polynomial-time algorithm that has two inputs

- an instance of a problem,
- and extra information,

and produces "Yes" if the inputs are a yes-instance and a polynomial-size certificate and "No" otherwise.

- here must be a certificate for each yes-instance, but not for each no-instance

NP Complexity Class

NP is the class of **decision problems** that can be **verified** in polynomial time using a polynomial-size certificate.

Recipe for Membership in NP

1. Give a certificate for each yes-instance and show that its size is at most polynomial.
2. Give a verification algorithm and show that its worst-case running time is at most polynomial.

3. Show that the algorithm answers "Yes" for any yes-instance and its certificate.
4. Show that the algorithm is not fooled by false certificates for any no-instances.

NP-Hardness and NP-Completeness

If every X $\in$ NP is reducible to Y, then Y is **NP-hard**.

If Y is in NP and Y is NP-hard, then Y is **NP-complete**.

Proving P != NP

If there exists even one NP-complete problem that is **not in P**, then NP contains a problem that is not in P, and so P ≠ NP.

Proving P = NP

If there exists even one NP-complete problem that is in P, then every problem in NP is in P, and so P = NP

Hard A, Hard B

If A can be reduced to B and A is "hard", then B must be "hard" too.

- Since a reduction from A to B means that A is "no harder" than B, we can show that if A is "hard", then B must be "hard" too
- This however, DOES NOT imply "Hard B, Hard A"
  - You cannot conclude that B being hard implies A being hard. (Since A can be an algorithm that does not use B)

Proving NP-Hardness

Based on the statement that "If Y can be reduced to Z and Y is NP-hard, then Z must be NP-hard too", to prove that Z is NP-hard:

1. Find a single problem Y that is NP-hard. (Identify properties of a generic x in S .)
2. Find a reduction from Y to Z. (Show that $P(x)$ is true, P(x) = x is reducible to Z)

Proving NP-Completeness

1. Prove Z is in NP.
2. Select Y that is known to be NP-complete.
3. Give an algorithm to compute a function f mapping each instance of Y to an instance of Z (it needn't map to all of Z)
4. Prove that if x is a yes-instance for Y then f(x) is a yes-instance for Z.
5. Prove that if f(x) is a yes-instance for Z then x is a yes-instance for Y.

6. Prove that the algorithm computing f runs in at most polynomial time.

Caution Regarding the Proof

- Make sure the reduction is in the right direction.
- Be sure to ensure that f maps all instances of Y.
- Step 5 should not just be a repetition of Step 4.
- Don't believe Step 3 without checking Steps 4 and 5. (You might need a few attempts to get Step 3 right.)
- Don't forget Step 6. (It is easy to ignore after all the hard work of the other steps.)

Module 8: Compromising on Time

Because NP-complete problems are unlikely to be able to be solved using algorithms that run in polynomial time, we need to compromise either on time, as we will discuss in this module, or on correctness, which we'll discuss in the next module.

Improving on Exhaustive Search

Goals:

- Generate only useful possibilities by checking them as we build them up.
- Ensure that we do not miss any correct solutions by making sure that no useful possibilities are missed.

We can describe each group by using a **partial solution**, a partial specification that corresponds to the set of all solutions consistent with the information specified.

The process of dividing a group into smaller groups corresponds to the process of **extending a partial solution** in all possible ways to form a larger partial solution, if possible.

A partial solution is a **candidate** if it is of the correct form to be a solution or witness for the problem.

- A candidate needs to have the correct form to possibly be a solution, but is not required to be a solution.

To make sure that we are not losing any candidates, the union of all the candidates associated with the extensions of a partial solution will equal the set of candidates associated with the partial solution.

Search Trees

Instead of constructing the tree and then searching it, we construct the tree **implicitly** by using a recursive procedure to explore the partial solutions.

- Each node in the tree corresponds to a function application of a recursive function.
- Often there is one non-recursive function used at the root of the tree, and a recursive function used at each of the other nodes.

At an internal node, the subtree rooted at the first child is explored completely before the subtree rooted at the second child.

- Each base case of a recursive function corresponds to a leaf in the implicit search tree, either a candidate or a partial solution that cannot be extended.
- When a base case has been reached, the search backtracks up the tree to find the closest node that was not completely explored.

Backtracking

The paradigm of backtracking makes use of a search tree to solve a problem.

Sketch

- Start with the initial partial solution at the root
- Initialize extra information
- At the current node:
  - Stop if the partial solution is a candidate
  - Stop if the partial solution cannot be extended
  - Recursively process each child in order
  - Update the extra information
  - Update the output
- Backtrack

Checlist

- Definition of a partial solution
- Definition of a partial solution at the root
- Definition of extra information
- Process for the root
- Process for a non-root
- Process for determining that a partial solution is a candidate
- Process for determining that a partial solution cannot be extended
- Process for extending a partial solution
- Process for updating the extra information
- Process for determining the output

Problem Types

- Decision Problems

- Search Problems
- Enumeration Problems
- Optimization Problems

The running time will be bounded above by the product of the **number of nodes** explored and the **worst-case cost of exploring a node**. To reduce the number of nodes searched:

- **Pruning**: Stop search as soon as it is clear that the partial solution cannot lead to a solution.
- **Comparing to best-so-far**: Stop search when a partial solution cannot lead to a solution better than the best found so far.
- **Choosing an ordering**: Avoid exploring duplicate solutions and/or explore more likely candidates first.

Branch-and-Bound

The paradigm of branch-and-bound makes use of a **bounding function**, which is defined as follows:

- for a **maximization problem**, the bounding function determines an **upper bound** on the value of any candidate formed by extending the current partial solution, and
- for a **minimization problem**, the bounding function determines a **lower bound** on the value of any candidate formed by extending the current partial solution.
- for this course, we will assume that the order in which nodes are explored is the same as in backtracking

Sketch

- Start with the initial partial solution at the root
- Initialize extra information
- At the current node:
  - Stop if the partial solution is a candidate
  - Stop if the partial solution cannot be extended
  - **Stop if the bound is worse than the best-so-far**
  - Recursively process each child in order
  - Update the extra information
  - Update the output
- Backtrack

Checklist

- Definition of partial solution
- Definition of partial solution at the root
- Definition of extra information

- **Definition of a bounding function**
- Process for the root
- Process for a non-root
- Process for determining that a partial solution is a candidate
- **Process for determining that a partial solution cannot be extended**
- Process for extending a partial solution
- Process for updating the extra information
- Process for determining the output

Summary of Backtracking and Branch-and-Bound

Properties

- Both algorithms depend on the exploration of an implicit search tree, using various methods to try to reduce how much of the tree needs to be examined.
- Backtracking can be used on any problem.
- Correctness for backtracking and branch-and-bound depends on making sure that any pruning is of parts of the tree that cannot contain solutions.
- Running time can be prohibitively high.
    - Constructing solutions more than once, resulting in behaviour worse than exhaustive search

Types of Outputs

- Ordering of elements
- Arrangement of elements
- Subset of elements
- Partition of elements

Module 9: Compromising on Correctness

Approximation Algorithms

An **approximation algorithm** is an algorithm for an optimization problem with guarantees concerning the approximate solution it produces.

- The approximate solution is a feasible solution.
- There is a provable **bound** on how close the approximate solution is to the optimal solution.
    - can be expressed as a **ratio** A/O, where A is the approximate solution and O is the optimal solution
    - For a **minimization** problem, we want A/O as small as possible (the upper bound on A)

    - note that O/A <= 1
- For a maximization problem, we want O/A as small as possible (the lower bound on A)
    - note that A/O <= 1

Ratio Bounds

An approximation algorithm has a **ratio bound** of $R(n)$ if for A, the value of the approximate solution, and O, the value of the optimal solution for any instance of size n, $\max\{A/O, O/A\} \leq R(n)$

- A ratio bound gives an **upper bound** on how bad the output of an approximation algorithm can be compared to the optimal solution. The bound will hold for any input to the algorithm.
- For a **minimization problem**, the second term is less than one, so we focus on the **first term (A/O)**.
- For a **maximization problem**, the first term is less than one, so we focus on the **second term (O/A)**.

Disproving a ratio bound

- Choose an instance x of size n .
- Show that x is an instance of the problem the algorithm is supposed to solve.
- Show that $\max\{A/O, O/A\} > R(n)$.

Disproving a ratio bound for $\Theta(f(n))$

- Choose a family of instances (for arbitrarily large, not necessarily every value, n).
- Show that each instance is an instance of the problem the algorithm is supposed to solve.
- Show that $\max\{A/O, O/A\} > g(n)$ where $g(n) \geq f(n)$ and $g(n) \notin \Theta(f(n))$.

If the ratio bound of an algorithm is a constant (that is, **independent of the size of the instance**), then the algorithm has a **constant ratio bound**

- If a problem can be solved by a polynomial-time approximation algorithm with a constant ratio bound, then the problem is in the **complexity class APX**.

How can we compared O and A if we do not know the optimal solution?

- One way of proving a ratio bound is to relate both O and A to a **third value M**. If we can find bounds on O and A in terms of M , then we can find an upper bound on $\max\{A/O, O/A\}$.

Types of Bounds

- For a **maximization problem**, we find a **lower bound** on the **approximate solution** and an **upper bound** on the **optimal solution**.
- For a **minimization problem**, we instead find an **upper bound** the **approximate solution** and a **lower bound** on the **optimal solution**.

Analyzing an Approximation Algorithm

1. Choose a **new measure** or problem.
2. Bound the approximate solution in terms of the new measure or problem.
3. Bound the optimal solution in terms of the new measure or problem.
4. Combine the two results to relate the approximate and optimal solutions.

Inapproximability

An inapproximability result shows that if there exists a certain type of approximation algorithm for a problem, then P = NP. Since it isn't likely that P = NP, such a result is strong evidence that no such approximation algorithm exists.

- If TSP is in APX, then P=NP.

Heuristics

The search for a solution to an optimization problem can be viewed as the exploration of the **search space** of feasible solutions.

- Another way to visualize a search space is by plotting the value of each feasible solution on a vertical axis.

Hill Climbing

- For a maximizaation problem, the idea behind hill climbing is that at each step, we move from the current feasible solution "up a hill" to a better feasible solution.
  - similar procedure can also be followed for a minimization problem
- Characteristics:
  - Start with an arbitrary feasible solution.
  - Repeatedly make a small change to improve the feasible solution.
  - Stop when no more improvements can be made.
- This is a known as a **local search heuristic** as we are looking at changes that move us from a feasible solution to another one that is "nearby" in the search space.
  - The key to the strategy is that we make a series of small improvements so that each new feasible solution is not too different from the previous one.
- Limitations:
  - Moving up a hill to reach its peak may lead to a **local maximum** (a point with no

higher "neighbour"), but a local maximum is not guaranteed to be the **global maximum**.

Other Heuristics

- Choose an initial solution judiciously instead of starting with an arbitrary solution.
- Use **steepest ascent**, which chooses the option that gives the **best improvement**.

Properties

- Hill-climbing makes changes in a feasible solution to try to construct a better feasible solution.
- Typically, heuristics are used on optimization problems.
- There is no guarantee that the solution is correct.
- Depending on how the heuristic is constructed, there may or may not be a guaranteed bound on running time.

Module 10: Changing the Rules

Assumptions in the Courses So Far

- We have no extra information about the instances we will be solving.
  - By changing this assumption, we look for special cases that make problems easier to solve
- Our algorithms are deterministic.
  - By changing this assumption, we add randomization to algorithms
- When problems are NP-complete, we need to compromise on running time or correctness.
  - By changing this assumption, we focus on one or more parameters of a problem that might be the key to the hardness of the problem
- We know the entire input at the start of the computation.
  - By changing this assumption, we see how to start solving a problem before all of the input is known.

Strongly and Weakly NP-Complete Problems; Special Instances

An NP-complete problem is

- **weakly NP-complete** if there is a known **pseudo-polynomial time algorithm** that solves the problem
  - we say that an algorithm runs in **pseudo-polynomial time** if the running time is polynomial in the largest integer in the input
- strongly NP-complete if it can be proved that it cannot be solved by a pseudo-

polynomial time algorithm unless P=NP.

We can solve an instance of a **weakly NP-complete problem** in polynomial time if the size of the largest integer used in the running time of the pseudo-polynomial time algorithm is at most polynomial in the size of the instance.

We may be able to solve an instance of an NP-complete problem in polynomial time for a certain class of instances.

Special Cases of graph

- If we restrict the number of neighbours any vertex can have, as in a **bounded degree graph** it might be possible to obtain a faster algorithm.
- In a **regular graph** each vertex has exactly the same number of neighbours; there are problems that are easier to solve on regular graphs than on graphs in general.
- there exist "treelike" graphs, known as **graphs of bounded treewidth**, which, although they are not necessarily trees, have some of the same useful properties.
- **planar graphs**, graphs that can be drawn on a piece of paper without any of the lines crossing, can be split into smaller planar graphs of size at most $2n/3$ by removal of a small number of vertices

Randomized Algorithms

All the algorithms we have seen so far are **deterministic**, which means that the steps taken by an algorithm on a specific instance are completely determined: if the algorithm is run repeatedly on the same instance, each time it runs, it will behave the same way.

- This type of behaviour is true even in when we used arbitrary choices when making decisions for greedy algorithms

In a randomized algorithm, **certain steps are chosen randomly**, in a manner equivalent to flipping a coin to decide what to do. Because different courses of action are possible, the same algorithm on the **same instance** may lead to **different running times** (ex: Las Vegas), or may lead to **different outputs** (Monte Carlo).

Running Time

- When randomly-chosen steps result in **different running times** for different executions of the same algorithm on the same instance, algorithms are assessed based on the running time, averaged over all possible executions.
    - The **expected running time** is not the same as the average case, which we used to discuss the behaviour of deterministic algorithms.
        - The average case is based on a probability distribution on the instances
        - To determine such a probability distribution, it is necessary to make

assumptions about the relative likelihood of the possible instances being used.
    - to calculate expected running time, we consider the average over possible executions on a **single instance**. There is no dependence on assumptions about distributions on instances.

Las Vegas Algorithm

A Las Vegas algorithm is guaranteed to give the **correct answer** in expected polynomial time.

- In general, calculating expected running times can be challenging. We will consider only a few simple examples in this course
- Sometimes, randamized your algorithm will lead to **faster running time**, but the algorithm will only work well if the randomized result is good.
    - if it is bad, make sure to randomly choose another one.

The running times of the algorithms depend on the probability of choosing a "good parameter", which results in a low running time

For randomized algorithms that you implement in this course, you will use the built-in `random` module.

Monte Carlo Algorithm

A Monte Carlo algorithm is guaranteed to **run in time polynomial** in the size of the input, and gives the correct answer with high probability.

For situations in which the randomness has an impact on the correctness of decision problems, we distinguish between **false positives** (the algorithm produces "yes" for a no-instance) and **false negatives** (the algorithm produces "no" for a yes-instance).

- For a decision problem, if there can be both false positive and false negatives, the error is considered to be **two-sided error**. If there can only be false positives, or if there can only be false negatives, the situation is considered to have **one-sided error**.

Reducing Errors

- For an algorithm with one-sided error, the probability of error will decrease as the **number of repetitions increases**. If there is a probability of error of p for each iteration, after k iterations the probability of error is p^k .
- For two-sided error, **repetition** can also be used to reduce the probability of error. To determine the answer, the algorithm will count the number of yeses and the number of nos obtained and return whatever appears more times.

Monte Carlo vs. Las Vegas

- If we have a Las Vegas algorithm, it is possible to convert it to a Monte Carlo algorithm.
  - we do this by sacrificing the correctness of Las Vegas algorithm
- There is no known general way to convert a Monte Carlo algorithm to a Las Vegas algorithm.

Pros and Cons of Randomized Algorithms

Pros

- With luck, it can be faster than a deterministic algorithm.
- A randomized algorithm may be simpler than a deterministic algorithm.
- Different answers on different runs may allow repeated runs to increase confidence in an answer.

Cons

- You may give up guarantees on worst-case time or correctness.
- Debugging is difficult, as there may be different outputs on different runs.
- Analysis is often difficult, such as the solving of probabilistic recurrence relations (outside the scope of this course).

Parameterized Algorithms

Parameterized Problems

We can define a problem using k as a parameter, forming a **parameterized problem**. Our goal is to create a function that is polynomial in terms of the instance size, but possibly much worse in terms of the parameter or parameters.

Parameterized Algorithms

A **fixed-parameter tractable algorithm** runs in time $O(g(p)n^{O(1)})$, where p is a parameter and g is any function

- The function g must be a function of p only, and not a function of n .

Bounded Search Tree

The bounded search tree paradigm makes use of a search tree, like in backtracking and branch-and-bound

- the worst-case cost of building and exploring a search tree can be bounded above by the product of the number of nodes in the tree and the cost of creating and exploring a single node
- the parameter p is used to determine when to stop searching

A tree of height h in which each node has at most c children has $O(c^h)$ nodes.

Kernelization

Kernelization is the process of replacing an instance by a **kernel**, a smaller instance of same problem. Our goal is to ensure the following properties:

1. the kernel is a yes-instance if and only if the original instance is a yes-instance, and
   - in order to solve the original instance, all we need to do is (a) form the kernel and then (b) solve the problem on the kernel
   - To obtain an algorithm that is fixed-parameter tractable, we need to make sure that (a) and (b) can both be accomplished by fixed-parameter tractable algorithms.
2. the size of the kernel is bounded by a function of p .
   - exponential in a function of p is still a function of p , and hence still fixed-parameter tractable

To use the kernelization approach, we'd like to show the following:

- Solving the problem on the kernel can be used to solve the original instance.
- The size of the kernel is bounded by a function of k .

Fixed-Parameter Complexity

A problem is in the complexity class **FPT** if there exists a fixed-parameter tractable algorithm that solves the problem.

Just like P is "easy" and NP-complete is "hard", for parameterized problems, FPT is "easy" and **W[1]-hard** is "hard". In fact, there is a hierarchy of "hard" classes, where W[i+1] is harder than W[i].

If there is a **parameterized reduction** from A to B and B is in FPT, then A is also in FPT.

- In such a reduction, the size of the parameter in B is a function of the size of the parameter in A , and the instance can be created in FPT time.
- the polynomial-time reduction does not ensure that the size of the parameter is bonded, and therefore may not be a parameterized reduction

Online Algorithms

Algorithms that have access to all of the input before starting computation are known as **offline algorithms**.

If instead the algorithm is required to start making decisions before all of the input has been provided, it is called an **online algorithm**.

Competitive Ratios

An online algorithm has **competitive ratio** c if the cost is *at most* $\boldsymbol{c}$ **times the cost of the best offline algorithm** for any input. The cost can be running time or other costs such as solution values.

Module 11: Extra Fun Stuff

Turing Machine

The significance of Turing machines lies in the **Church-Turing thesis**, which states that a function can be computed if and only if it can be computed using a Turing machine

Circuit Models

A Boolean circuit is formed from **gates**, where the output of a gate may be connected by a wire to an input of another gate.

Examples:

- OR: the output is a 1 if any input is a 1 and 0 otherwise
- AND: the output is a 1 if all inputs are 1's and 0 otherwise
- NOT: the output is a 1 if the input is a 0 and 1 otherwise

To determine the cost of an algorithm on a circuit, typical measures include the number of gates in a circuit (size of a circuit) and the maximum number of gates on the path from an input gate to the output gate (depth of a circuit).

It is also possible to form an **arithmetic circuit**, and the gates compute arithmetic operations

Types of Computation Machines

Parallel computation

- uses multiple machines that coordinate to split the work.
- Consider the following questions:
  - Can more than one machine read the same variable at the same time?
  - Can more than one machine write the same variable at the same time?
  - What happens if multiple machines write the same variable at the same time?

Distributed computation

- involves communication among multiple machines by passing messages asynchronously.

- A simple, but not easy, task in a distributed environment is **consensus**, for which the goal is to get all the processes to agree on a value, such that:
  - all processes that don't crash output the same value, and
  - the value is an input to one of the processes.

Quantum computation

- studies the power and limitations of quantum computers.
- Quantum computers use **quantum bits** that can have more complicated values than 0's and 1's. Information can be combined in quantum registers, made up of multiple quantum bits

Complex Relations Among Data Items

Data structures are various ways of organizing data in computer memory. Different arrangements lead to different costs for accessing and updating data. Efficient algorithms can rely on the choice of data structures that are used.

Numerial Data

for handling of very large numbers, arithmetic operations have a cost proportional to the space needed to store the value.

- Since the number of bits in a binary numbers is logarithmic in the size of the number , this model is known as a **log cost model**.

Handling Change

A variety of different types of change are handled in algorithms using terms such adaptive and dynamic.

Analyzing Algorithms

An **output sensitive** algorithm has a running time measured in terms of not only the size of the input but also the **size of the output**

For a sequence of independent tasks, we might use **amortized analysis**, which focuses on the cost of a worst-case sequence of tasks rather than an individual task in isolation.

Complexity for Time and Space

If instead we have a polynomial-time verification algorithm using a polynomial-size certificate for every no-instance, our problem is in **co-NP**

- A problem that is in the intersection of NP and co-NP, that is one for which there exist polynomial-size certificates for both yes-instances and no-instances, is in P

An entire hierarchy can be built by further generalizations of the classes P and NP, and can be defined using a type of Turing machine known as an **alternating Turing machine**, which allows the expression of universal and existential statements.

As you go up the **polynomial-time hierarchy**, classes of problems can be expressed using more and more alternations between universal and existential statements.

- The complex structure of classes in the hierarchy is a fragile one, however, since it is not known whether the classes are distinct. If P = NP, then the entire hierarchy collapses, with all classes in the hierarchy being equal to P.

The class **PSPACE** consists of problems that can be solved using a polynomial amount of space.

- Anything that is in P is in PSPACE
- In fact, every class in the polynomial-time hierarchy is contained in PSPACE

Complexity of Approximation Algorithms

An approximation algorithm has a relative error bound of $\epsilon(n)$ if $|A-O|/O \leq \epsilon(n)$, where A is the value of the solution obtained by the algorithm and O is the value of the optimal solution.

- Such an algorithm is called an **$\epsilon(n)$ -approximation algorithm**.

Given an algorithm that obtains a particular relative error bound, it may be possible to decrease the relative error bound by increasing the running time.

- When it is possible to do so for any chosen relative error bound, the way of forming such a family of approximation algorithms is known as an **approximation scheme**.

More formally, an approximation scheme can be viewed as an algorithm that produces algorithms. Such an algorithm takes as input an instance and an ϵ such that for any fixed ϵ , it produces an approximation algorithm with relative error bound ϵ .

- The algorithm is a **polytime approximation scheme** if for any fixed ϵ , the running time is polynomial in the size of the instance.
- The algorithm is a **fully polytime approximation scheme** if the running time is polynomial in both $1/\epsilon$ and n .

Compelxity Class of Approximation Algorithms

- **APX** contains any problem that can be solved by a polynomial-time approximation algorithm with constant ratio bound.

- **PTAS** contains any problem that can be solved by a polynomial-time approximation scheme.
- **FPTAS** contains any problem that can be solved by a fully polynomial-time approximation scheme

FPTAS ⊆ PTAS ⊆ APX

A problem is APX-hard if there is a PTAS-reduction from every problem in APX to that problem

Complexiy for Randomized Algorithms

- **RP** contains any problem that can be solved using a Monte Carlo algorithm with bounded one-sided error and polynomial running time.
- **BPP** contains any problem that can be solved using a Monte Carlo algorithm with bounded two-sided error and polynomial running time.
- **ZPP** contains any problem that can be solved using a Las Vegas algorithm with expected polynomial running time.

The classes can be related as follows: ZPP ⊆ RP ⊆ BPP

Sometimes a randomized algorithm can be **derandomized** to form a deterministic algorithm. One option is to make the number of random bits small enough that all possibilities can be tried in polynomial time, using exhaustive search.

We can form the following hierarchy: P ⊆ ZPP ⊆ RP ⊆ NP

You may have noticed that BPP is not listed here, as the relationship between BPP and NP is unknown. It is known that if P = NP, then P = BPP.

Complexity for Prallel computation

the class **NC** is used to classify "good" problems that can be solved in $O(\log^c(n))$ time using a polynomial number of processors.

- Since NC is considered "easy" and P is considered "hard" in this setting, we can define problems to be **P-hard** based on NC reductions.

Reductions and classes have also been defined with respect to space. The class **L** contains decision problems that can be solved deterministically in logarithmic space, and the class **NL** consists of decision problems that can be solved nondeterministically in logarithmic space.

The classes are related as follows: NC ⊆ L ⊆ NL ⊆ P. Whether NC is equal to P is another open question.

Computability

The area of computability categorizes problems that cannot be solved at all.