ECE 374 B: Algorithms and Models of Computation, Summer 2024
Final – August 02, 2024

- **You will have 120 minutes (2 hours) to solve all the problems. Most have multiple parts.** Don't spend too much time on questions you don't understand and focus on answering as much as you can!
- *No* resources are allowed for use during the exam except a multi-page cheatsheet and scratch paper on the back of the exam.
- Incorrect algorithms will receive a score of 0, but slower than necessary but correct algorithms will *always* receive some points, even brute force ones. Thus, *you should prioritize the correctness of your submitted algorithms over speed*; you will receive more points that way. On the other hand, submit the fastest algorithms that you know are correct; faster algorithms will receive more points.
- Any recursive backtracking algorithm or dynamic programming algorithm given without an *English* description of the recursive function (i.e., a description of the output of the function *in terms of their inputs*) will receive a score of 0.
- Any greedy algorithm or a modification of a standard graph algorithm given without a proof of correctness will receive a score of 0.
- Any algorithms written in actual code instead of pseudocode will receive a score of 0.
- For problems with a graph given as input, you may assume the graph is simple (i.e., it has no self-loops or parallel edges).
- Unless explicitly mentioned, **a runtime analysis is required for each given algorithm**.
- Unless otherwise stated, assume $\mathsf{P} \neq \mathsf{NP}$.
- Assume that whenever the word "reduction" is used, we mean a (not necessarily polynomial-time) *mapping/many-one* reduction.
- ***Don't cheat.*** If we catch you, you will get an F in the course.
- ***Good luck!***

Name: ____________________

NetID: ____________________

1 Short answer I (2 questions) - 12 points

For each of the following, choose the ***best*** answer, meaning if there are multiple correct answers and one of them implies the others, you should choose the one that implies the others.

1. Suppose your algorithm's runtime is given by the recurrence relation $T(n+1) = 4T(n) + 1$, assuming $T(0) = 1$. Give a **tight** *asymptotic* solution to this recurrence to summarize the running time. State whether this algorithm is *polynomial time* or not.

2. The n queens problem asks if we can fit n queens on a $n \times n$ chess board without attacking one-another. As we discussed in lecture, we definitely can assuming $n > 5$. But give a concise algorithm that finds a valid configuration of n queens that on the chessboard. Assume you have a blackbox ($O(1)$) function CheckQueens(config) that takes in a configuration of queens and returns true if the queens cannot attack one another (false otherwise). What is the runtime of your algorithm?

2 Short answer II (2 questions) - 12 points

For the short answers, keep solutions brief and concise. Don't need a long explanation on why the answer is correct. As long as your solution is correct, you're good.

1. What is the asymptotic running time of the Tower of Hanoi problem?

2. Let's say we program the Linear time selection algorithm, but instead of lists of size 5, we break the input array into lists of size 3.

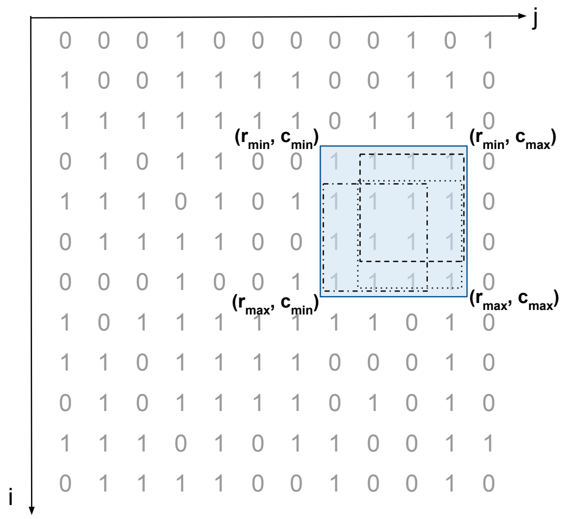
   (a) What is the recurrence that describes this new algorithm

   (b) What is the asymptotic bound of this recurrence?

3 Square - 14 points

Let $M[1..n, 1..n]$ be an $n \times n$ bitmap of 0s and 1s. A *square* in M has four endpoints $(r_{\min}, c_{\min}), (r_{\min}, c_{\max}), (r_{\max}, c_{\min}), (r_{\max}, c_{\max})$, where $1 \leq r_{\min} \leq r_{\max} \leq n$, $1 \leq c_{\min} \leq c_{\max} \leq n$ and $r_{\max} - r_{\min} = c_{\max} - c_{\min}$. Define $\ell := 1 + r_{\max} - r_{\min} = 1 + c_{\max} - c_{\min} > 0$ to be the *side length* of such a square.

Given an $n \times n$ bitmap as an array $M[1..n, 1..n]$ of 0s and 1s, describe and analyze an efficient algorithm to compute the ***maximum side length*** of a square ***containing only 1s*** in M. [1] For example, your algorithm should return 4 given the following bitmap M as input, as the shaded blue square is the square of 1s in M:



[1]Hint: Look at the dashed lines in figure above. A square is made up of multiple smaller squares.

4 Minimum Spanning trees - 14 points

Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

- Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.
- Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

In both cases, the input to your algorithm is the new edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree for the new graph. [Hint: Consider the cases $e \in T$ and $e \notin T$ separately.

5 Algorithmic Complexity I - 12 points

The short simple path SSP problem is asks if there is a short simple (can visit any vertices more than once) path between two nodes s and t of less than length k:

- INPUT: A directed graph that contains negative edge weights and possibly cycles (G) and the start (s) and end vertices (t) and an integer k.
- OUTPUT: TRUE if there exists a simple path $s \to t$ in G whose length is less than k.

Which of the following complexity classes does SSP belong to? Circle ***all*** that apply:

P NP co-NP NP-hard NP-complete

Explain your choice(s):

6 Algorithmic Complexity II - 12 points

A triangle in an undirected graph is a clique of three vertices. The TRIANGLE problem is defined as follows:

- INPUT: An ***undirected*** graph G.
- OUTPUT: TRUE if G contains a complete graph on ***three*** vertices as a subgraph, and FALSE otherwise.

Which of the following complexity classes does TRIANGLE belong to? Circle ***all*** that apply:

P NP co-NP NP-hard NP-complete

Explain your choice(s):

7 Algorithmic Complexity III - 12 points

The JustNeedOne-3SAT problem asks whether it is possible to satisfy at least one clause and is defined as follows:

- Input: A 3SAT formula ϕ.
- Output: True if ϕ has a variable assignment that satisfies at least one clause and False otherwise.

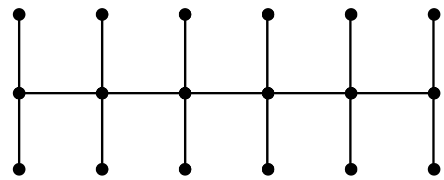
For the sake of simplicity, lets assume the number of clauses and variables is large (> 100), (this is just to avoid annoying edge cases). Which of the following complexity classes does JustNeedOne-3SAT belong to? Circle ***all*** that apply:

P NP NP-hard NP-complete

Explain your choice(s):

8 Algorithmic Complexity IV - 12 points

A centipede is an undirected graph formed by a path of length k with two edges (legs) attached to each node on the path as shown in the below figure. Hence, the centipede graph has 3k vertices.



CENTIPEDE graph with $k = 6$

The CENTIPEDE problem asks whether there exists a centipede subgraph of $3k$ vertices :

- INPUT: A undirected graph $G = V, E$ and integer k
- OUTPUT: TRUE if there does exist a CENTIPEDE graph. False otherwise.

Which of the following complexity classes does CENTIPEDE belong to? Circle ***all*** that apply:

P NP NP-hard NP-complete

Explain your choice(s):

This page is for additional scratch work!

ECE 374 B Complete: Cheatsheet

1 Languages and strings

Languages

Definitions

- An *alphabet* Σ is a **finite** set of symbols.
- A *string* in Σ^* is a **finite** sequence of symbols in Σ.
- A *language* is L is a set of strings over some alphabet.

All languages represent mathematical problems.
Example: multiplication of two integers:

$$L_{MULT2} = \begin{Bmatrix} 1\times 1|1, & 1\times 2|2, & 1\times 3|3, \dots \\ 2\times 1|2, & 2\times 2|4, & 2\times 3|6, \dots \\ \vdots & \vdots & \vdots \\ n\times 1|n, & n\times 2|2n, & n\times 3|3n, \dots \end{Bmatrix} \tag{1}$$

Language operations

- For languages A, B the *concatenation* of A, B is $AB = \{xy \mid x \in A, y \in B\}$.
- For languages A, B, their *union* is $A \cup B$, *intersection* is $A \cap B$, and *difference* is $A \setminus B$ (also written as $A - B$).
- For language $A \subseteq \Sigma^*$ the *complement* of A is $\bar{A} = \Sigma^* \setminus A$.
- Σ^n is the set of all strings of length n.
- $\Sigma^* = \cup_{n\geq 0}\Sigma^n$ is the set of all strings over Σ.
- $\Sigma^+ = \cup_{n\geq 1}\Sigma^n$ is the set of non-empty strings over Σ.

Strings

Definitions

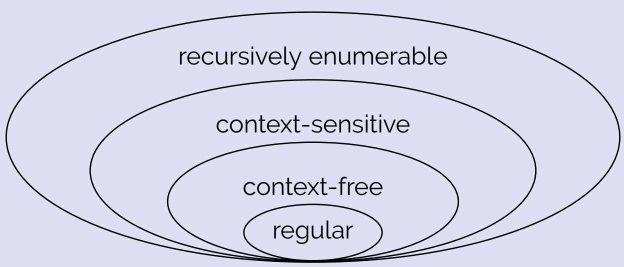
- The *length* of a string w (denoted by $|w|$) is the number of symbols in w.
- For integer $n \geq 0$, Σ^n is set of all strings over Σ of length n. Σ^* is the set of all strings over Σ.
- Σ^* is the set of all strings of all lengths including empty string.
- ε is a *string* containing no symbols.
- $\varnothing$ is the *empty set*. It contains no strings.

String operations

- If x and y are strings then xy denotes their concatenation. Recursively:
  - $xy = y$ if $x = \varepsilon$
  - $xy = a(wy)$ if $x = aw$
- v is *substring* of w $\iff$ there exist strings x, y such that $w = xvy$.
  - If $x = \varepsilon$ then v is a *prefix* of w
  - If $y = \varepsilon$ then v is a *suffix* of w
- A *subsequence* of a string $w = w_1w_2\dots w_n$ is either a subsequence of $w_2\dots w_n$ or w_1 followed by a subsequence of $w_2\dots w_n$.
- If w is a string then w^n is defined inductively as follows: $w^n = \varepsilon$ if $n = 0$ or $w^n = ww^{n-1}$ if $n > 0$

2 Overview of language complexity

Overview



Grammar	Languages	Production Rules	Automaton	Examples
Type-0	recursively enumerable	$\gamma \to \alpha$ (no constraints)	Turing machine	$L = \{w \mid w$ is a *TM* which halts$\}$
Type-1	context-sensitive	$\alpha A\beta \to \alpha\gamma\beta$	linear bounded nondeterministic Turing machine	$L = \{a^nb^nc^n \mid n > 0\}$
Type-2	context-free	$A \to \alpha$	nondeterministic pushdown automata	$L = \{a^nb^n \mid n > 0\}$
Type-3	regular	$A \to aB$	finite state machine	$L = \{a^n \mid n > 0\}$

Meaning of symbols:

- a - terminal
- A, B - variables
- α, β, γ - strings in $\{a \cup A\}^*$ where α, β are maybe empty, γ is never empty

[a]

[a] Table borrowed from Wikipedia: `https://en.wikipedia.org/wiki/Chomsky_hierarchy`

3 Regular languages

Regular language - overview

A language is regular if and only if it can be obtained from finite languages by applying

- union,
- concatenation or
- Kleene star

finitely many times. All regular languages are representable by regular grammars, DFAs, NFAs and regular expressions.

Regular expressions

Useful shorthand to denotes a language.
A *regular expression* $\mathbf{r}$ over an alphabet Σ is one of the following:
Base cases:

- $\varnothing$ the language $\varnothing$
- ε denotes the language $\{\varepsilon\}$
- a denote the language $\{a\}$

Inductive cases: If $\mathbf{r_1}$ and $\mathbf{r_2}$ are regular expressions denoting languages L_1 and L_2 respectively (i.e.,$L(\mathbf{r_1}) = L_1$ and $L(\mathbf{r_2}) = L_2$) then,

- $\mathbf{r_1} + \mathbf{r_2}$ denotes the language $L_1 \cup L_2$
- $\mathbf{r_1} \bullet \mathbf{r_2}$ denotes the language $L_1 L_2$
- $\mathbf{r_1^*}$ denotes the language L_1^*

Examples:

- 0^* - the set of all strings of 0s, including the empty string
- $(00000)^*$ - set of all strings of 0s with length a multiple of 5
- $(0+1)^*$ - set of all binary strings

Nondeterministic finite automata

NFAs are similar to DFAs, but may have more than one transition destination for a given state/character pair.

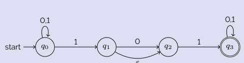
An NFA N *accepts a string* w iff some accepting state is reached by N from the start state on input w.

The *language accepted (or recognized)* by an NFA N is denoted $L(N)$ and defined as $L(N) = \{w \mid N \text{ accepts } w\}$.

A *nondeterministic finite automaton (NFA)* $N = (Q, \Sigma, s, A, \delta)$ is a five tuple where

- Q is a finite set whose elements are called *states*
- Σ is a finite set called the *input alphabet*
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \to \mathcal{P}(Q)$ is the *transition function* (here $\mathcal{P}(Q)$ is the power set of Q)
- s and Σ are the same as in DFAs

Example:



- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- δ :

	ε	0	1
q_0	$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_1, q_2\}$	$\{q_2\}$	$\varnothing$
q_2	$\{q_2\}$	$\varnothing$	$\{q_3\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$

- $s = q_0$
- $A = \{q_3\}$

For NFA $N = (Q, \Sigma, \delta, s, A)$ and $q \in Q$, the ε-reach(q) is the set of all states that q can reach using only ε-transitions.
Inductive definition of $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$:

- if $w = \varepsilon$, $\delta^*(q, w) = \varepsilon$-reach$(q)$
- if $w = a$ for $a \in \Sigma$, $\quad \delta^*(q, a) = \varepsilon\text{reach}\Big(\bigcup_{p \in \varepsilon\text{-reach}(q)} \delta(p, a)\Big)$
- if $w = ax$ for $a \in \Sigma, x \in \Sigma^*$: $\delta^*(q, w) = \varepsilon\text{reach}\Big(\bigcup_{p \in \varepsilon\text{-reach}(q)} \Big(\bigcup_{r \in \delta^*(p,a)} \delta^*(r, x)\Big)\Big)$

Regular closure

Regular languages are closed under union, intersection, complement, difference, reversal, Kleene star, concatenation, etc.

Deterministic finite automata

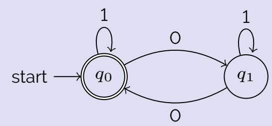
DFAs are finite state machines that can be represented as a directed graph or in terms of a tuple.

The *language accepted* (or recognized) by a DFA M is denoted by $L(M)$ and defined as $L(M) = \{w \mid M \text{ accepts } w\}$.

A *deterministic finite automaton (DFA)* $M = (Q, \Sigma, s, A, \delta)$ is a five tuple where

- Q is a finite set whose elements are called *states*
- Σ is a finite set called the *input alphabet*
- $\delta : Q \times \Sigma \to Q$ is the *transition function*
- $s \in Q$ is the *start state*
- $A \subseteq Q$ is the set of *accepting/final* states

Example:



- $Q = \{q_0, q_1\}$
- $\Sigma = \{0, 1\}$
- δ :

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

- $s = q_0$
- $A = \{q_0\}$

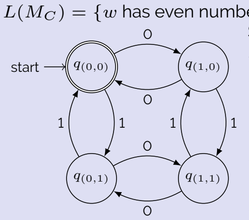
Every string has a unique walk along a DFA. We define the extended transition function as $\delta^* : Q \times \Sigma^* \to Q$ defined inductively as follows:

- $\delta^*(q, w) = q$ if $w = \varepsilon$
- $\delta^*(q, w) = \delta^*(\delta(q, a), x)$ if $w = ax$.

Can create a larger DFA from multiple smaller DFAs. Suppose

- $L(M_0) = \{w \text{ has an even number of 0s}\}$ (pictured above) and
- $L(M_1) = \{w \text{ has an even number of 1s}\}$.

$L(M_C) = \{w \text{ has even number of 0s and 1s}\}$

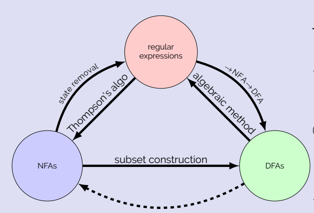


Suppose $M_0 = (Q_0, \Sigma, s_0, A_0, \delta_0)$ and $M_1 = (Q_1, \Sigma, s_1, A_1, \delta_1)$. Then

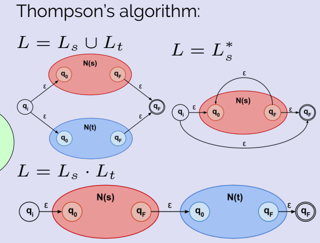
- $Q = Q_0 \times Q_1 = \{(q_0, q_1) \mid q_0 \in Q_0, q_1 \in Q_1\}$
- $s = (s_0, s_1)$
- $\delta : Q \times \Sigma \to Q$, where $\delta((q_0, q_1), a) = (\delta_0(q_0, a), \delta_1(q_1, a))$
- $A = \{(q_0, q_1) \mid q_0 \in A_0 \text{ and } q_1 \in A_1\}$

Regular language equivalences

A regular language can be represented by a regular expression, regular grammar, DFA and NFA.



Thompson's algorithm:



Arden's rule: If $R = Q + RP$ then $R = QP^*$.

Fooling sets

Some languages are not regular (Ex. $L = \{0^n 1^n \mid n \geq 0\}$).

Two states $p, q \in Q$ are *distinguishable* if there exists a string $w \in \Sigma^*$, such that

$\delta^*(p, w) \in A$ and $\delta^*(q, w) \notin A$.

or

$\delta^*(p, w) \notin A$ and $\delta^*(q, w) \in A$.

Two states $p, q \in Q$ are *equivalent* if for all strings $w \in \Sigma^*$, we have that

$\delta^*(p, w) \in A \iff \delta^*(q, w) \in A$.

For a language L over Σ a set of strings F (could be infinite) is a *fooling set* or *distinguishing set* for L if every two distinct strings $x, y \in F$ are distinguishable.

4 Context-free languages

Context-free languages

A language is context-free if it can be generated by a context-free grammar. A context-free grammar is a quadruple $G = (V, T, P, S)$

- V is a finite set of *nonterminal (variable) symbols*
- T is a finite set of *terminal symbols* (alphabet)
- P is a finite set of *productions*, each of the form $A \to \alpha$ where $A \in V$ and α is a string in $(V \cup T)^*$ Formally, $P \subseteq V \times (V \cup T)^*$.
- $S \in V$ is the *start symbol*

Example: $L = \{ww^R | w \in \{0,1\}^*\}$ is described by $G = (V, T, P, S)$ where V, T, P and S are defined as follows:

- $V = \{S\}$
- $T = \{0, 1\}$
- $P = \{S \to \varepsilon \mid 0S0 \mid 1S1\}$
  (abbreviation for $S \to \varepsilon, S \to 0S0, S \to 1S1$)
- $S = S$

Pushdown automata

A pushdown automaton is an NFA with a stack.

The language $L = \{0^n 1^n \mid n \geq 0\}$ is recognized by the pushdown automaton:

A *nondeterministic pushdown automaton (PDA)* $P = (Q, \Sigma, \Gamma, \delta, s, A)$ is a **six** tuple where

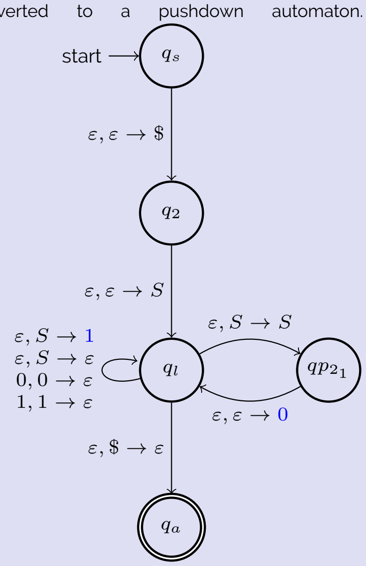
- Q is a finite set whose elements are called *states*
- Σ is a finite set called the *input alphabet*
- Γ is a finite set called the *stack alphabet*
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$ is the *transition function*
- s is the start state
- A is the set of accepting states

In the graphical representation of a PDA, transitions are typically written as $\langle$input read$\rangle$, $\langle$stack pop$\rangle$ $\to$ $\langle$stack push$\rangle$.

A CFG can be converted to a pushdown automaton.

The PDA to the right recognizes the language described by the following grammar:

$S \to 0S|1|\varepsilon$



Context-free closure

Context-free languages are closed under union, concatenation, and Kleene star.

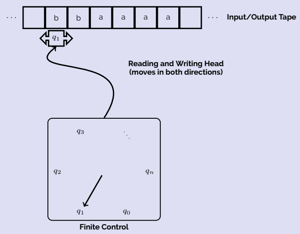
They are **not** closed under intersection or complement.

5 Recursively enumerable languages

Turing Machines

Turing machine is the simplest model of computation.

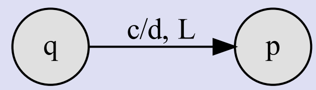
- Input written on (infinite) one sided tape.
- Special blank characters.
- Finite state control (similar to DFA).
- Ever step: Read character under head, write character out, move the head right or left (or stay).
- Every TM **M** can be encoded as a string $\langle M \rangle$



Transition Function: $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow, \square\}$

$\delta(q, c) = (p, d, \leftarrow)$

- q: current state.
- c: character under tape head.
- p: new state.
- d: character to write under tape head
- $\leftarrow$: Move tape head left.



6 Recursion

Simple recursion

Definitions

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

Tower of Hanoi

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

Pseudocode: Tower of Hanoi

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

Recurrences

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

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

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

Some useful identities:

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

Backtracking

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

Pseudocode: LIS - Naive enumeration

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

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

Pseudocode: LIS - Backtracking

```
LIS_smaller(A[1..n], x):
  if n = 0 then return 0
  max = LIS_smaller(A[1..n − 1], x)
  if A[n] < x then
    max = max {max, 1 + LIS_smaller(A[1..(n − 1)], A[n])}
  return max
```

Divide and conquer

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

Sorting algorithms

Algorithm	Runtime	Space
Mergesort	$O(n \log n)$	$O(n \log n)$ $O(n)$ (if optimized)
Quicksort	$O(n^2)$ $O(n \log n)$ if using MoM	$O(n)$

Karatsuba's algorithm

We can divide and conquer multiplication like so:

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

We can rewrite the equation as:

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

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

Linear time selection

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

Pseudocode: Quickselect with median of medians

```
Median-of-medians (A, i):
  sublists = [A[j:j+5] for j ← 0, 5, . . . , len(A)]
  medians = [sorted (sublist)[len (sublist)/2]
        for sublist ∈ sublists]

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

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

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

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

Dynamic programming

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

Longest increasing subsequence

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

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

Pseudocode: LIS - DP

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

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

Edit distance

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

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

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

Pseudocode: Edit distance - DP

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

  for $i = 1$ to m **do**
    for $j = 1$ to n **do**

$$M[i][j] = \min \begin{cases} COST\,[A[i]]\,[B[j]] \\ \quad + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

7 Graph algorithms

Graph basics

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

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

Directed acyclic graphs

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

Pseudocode: Kahn's algorithm

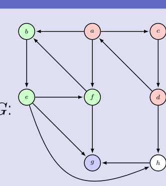
Kahn($G(V, E)$,u):
toposort$\leftarrow$empty list
for $v \in V$:
  in(v) $\leftarrow |\{u \mid u \to v \in E\}|$
while $v \in V$ that has in(v) $= 0$:
  Add v to end of toposort
  Remove v from V
  for v in $u \to v \in E$:
    in(v) $\leftarrow$ in(v) $- 1$
return toposort

Running time: $O(n+m)$

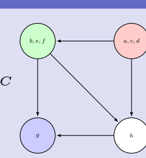
- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-visit order.

Strongly connected components

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



G^{SCC}



Pseudocode: Metagraph - linear time

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

DFS and BFS

Pseudocode: Explore (DFS/BFS)

```
Explore(G,u):
  for i ← 1 to n:
    Visited[i] ← False
  Add u to ToExplore and to S
  Visited[u] ← True
  Make tree T with root as u
  while B is non-empty do
    Remove node x from B
    for each edge (x, y) in Adj(x) do
      if Visited[y] = False
        Visited[y] ← True
        Add y to B, S, T (with x as parent)
```

Note:

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

Pre/post numbering

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

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

Minimum Spanning Tress

Some notes on minimum spanning trees:

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Tree = a connected graph with no cycles.
- Sub-graph H of G is *spanning* for G, if G and H have same connected components.
- A minimum spanning tree is composed of all the safe edges in the graph
- An edge $e = (u, v)$ is a *safe* edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).
- An edge $e = (u, v)$ is an *unsafe* edge if there is some cycle C such that e is the unique maximum cost edge in C.
- All edges are safe or unsafe.

Pseudocode: Boruvka's algorithm: $O(m \log(n))$

```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
  X ← ∅
  for each connected component S of T do
    add to X the cheapest edge between S and V \ S
  Add edges in X to T
return the set T
```

Pseudocode: Kruskal's algorithm: $(m + n)\log(m)$ (using Union-Find structure)

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty do
  pick e = (u, v) ∈ E of minimum cost
  if u and v belong to different sets
    add e to T
    merge the sets containing u and v
return the set T
```

Pseudocode: Prim's algorithm: $(n)\log(n) + m$ (using Priority Queue)

```
T ← ∅, S ← ∅, s ← 1
∀v ∈ V(G) : d(v) ← ∞, p(v) ← ∅
d(s) ← 0
while S ≠ V do
  v = arg min_{u∈V\S} d(u)
  T = T ∪ {vp(v)}
  S = S ∪ {v}
  for each u in Adj(v) do
    d(u) ← min { d(u)
                 c(vu)
    if d(u) = c(vu) then
      p(u) ← v
return T
```

Shortest paths

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

Pseudocode: Dijkstra

```
for v ∈ V do
  d(v) ← ∞
X ← ∅
d(s, s) ← 0
for i ← 1 to n do
  v ← arg min_{u∈V−X} d(u)
  X = X ∪ {v}
  for u in Adj(v) do
    d(u) ← min {(d(u), d(v) + ℓ(v, u))}
return d
```

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

Bellman-Ford algorithm:

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

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

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

Pseudocode: Bellman-Ford

```
for each v ∈ V do
  d(v) ← ∞
d(s) ← 0

for k ← 1 to n − 1 do
  for each v ∈ V do
    for each edge (u, v) ∈ in(v) do
      d(v) ← min{d(v), d(u) + ℓ(u, v)}

return d
```

Running time: $O(nm)$

Floyd-Warshall algorithm:

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

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

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

Pseudocode: Floyd-Warshall

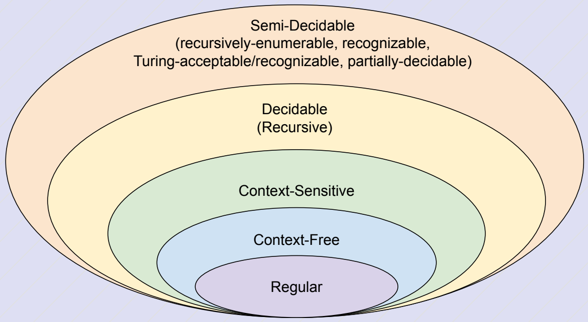
```
Metagraph(G(V, E)):
  for i ∈ V do
    for j ∈ V do
      d(i, j, 0) ← ℓ(i, j)
        (* ℓ(i, j) ← ∞ if (i, j) ∉ E, 0 if i = j *)

  for k ← 0 to n − 1 do
    for i ∈ V do
      for j ∈ V do
        d(i, j, k) ← min { d(i, j, k − 1),
                           d(i, k, k − 1) + d(k, j, k − 1)
  for v ∈ V do
    if d(i, i, n − 1) < 0 then
      return "∃ negative cycle in G"

  return d(·, ·, n − 1)
```

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

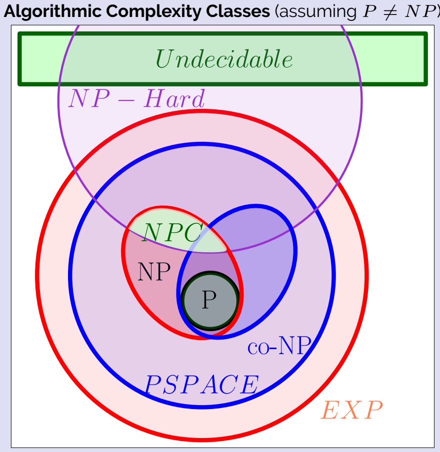
Complexity Classes

Computational Complexity Classes

Turing-unrecognizable
(everything outside of the complexity classes below)



Algorithmic Complexity Classes (assuming $P \neq NP$)



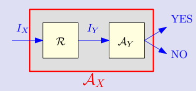
Reductions

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \implies Y$. But we know X is hard so Y has to be hard too. On the other hand if we know Y is easy, then X has to be easy too.

The Karp reduction, $X \leq_P Y$ suggests that there is a polynomial time reduction from X to Y.



Assuming

- $R(n)$: running time of $\mathcal{R}$
- $Q(n)$: running time of $\mathcal{A}_Y$

Running time of $\mathcal{A}_X$ is $O(Q(R(n)))$

Sample NP-complete problems

CircuitSat:	Given a boolean circuit, are there any input values that make the circuit output True?
3Sat:	Given a boolean formula in conjunctive normal form, with exactly three distinct literals per clause, does the formula have a satisfying assignment?
IndependentSet:	Given an undirected graph G and integer k, what is there a subset of vertices $\geq k$ in G that have no edges among them?
Clique:	Given an undirected graph G and integer k, is there a complete complete subgraph of G with more than k vertices?
kPartition:	Given a set X of kn positive integers and an integer k, can X be partitioned into n, k-element subsets, all with the same sum?
3Color:	Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?
HamiltonianPath:	Given graph G (either directed or undirected), is there a path in G that visits every vertex exactly once?
HamiltonianCycle:	Given graph G (either directed or undirected), is there a cycle in G that visits every vertex exactly once?
LongestPath:	Given a graph G (either directed or undirected, possibly with weighted edges) and an integer k, does G have a path $\geq k$ length?

- Remember a **path** is a sequence of distinct vertices $[v_1, v_2, \ldots v_k]$ such that an edge exists between any two vertices in the sequence. A **cycle** is the same with the addition of a edge $(v_k, v_1) \in E$. A **walk** is a path except the vertices can be repeated.
- A formula is in conjunction normal form if variables are or'ed together inside a clause and then clauses are and'ed together: $((x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_4 \vee x_5))$. Disjunctive normal form is the opposite $((x_1 \wedge x_2 \wedge x_3) \vee (\overline{x_2} \wedge x_4 \wedge x_5))$.

Sample undecidable problems

AcceptOnInput:	$A_{TM} = \{\langle M, w\rangle \mid M$ is a TM and M accepts on $w\}$
HaltsOnInput:	$Halt_{TM} = \{\langle M, w\rangle \mid M$ is a TM and halts on input $w\}$
HaltOnBlank:	$HaltB_{TM} = \{\langle M\rangle \mid M$ is a TM & M halts on blank input$\}$
Emptiness:	$E_{TM} = \{\langle M\rangle \mid M$ is a TM and $L(M) = \varnothing\}$
Equality:	$EQ_{TM} = \left\{\langle M_A, M_B\rangle \,\middle|\, \begin{array}{l} M_A \text{ and } M_B \text{ are TM's} \\ \text{and } L(M_A) = L(M_B) \end{array}\right\}$