1 Languages and strings

Languages

- An alphabet $\Sigma$ is a finite set of symbols.

Definitions

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

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

$$L_{MULT} = \{ 1 \times 1|1, \ 1 \times 2|2, \ 1 \times 3|3, \ldots \} \cup \{ 2 \times 1|2, \ 2 \times 2|4, \ 2 \times 3|6, \ldots \} \cup \{ \ldots \} \cup \{ n \times 1|n, \ n \times 2|2n, \ n \times 3|3n, \ldots \}$$

- 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 $\bar{A} = \Sigma^* \setminus A$.
- $\Sigma^+$ is the set of all strings of length $n$.
- $\Sigma^* = \cup_{n \geq 0} \Sigma^n$ is the set of all strings over $\Sigma$.
- $\Sigma_+ = \cup_{n \geq 1} \Sigma^n$ is the set of non-empty strings over $\Sigma$.

Language operations

- $\epsilon$ is a string containing no symbols.
- $\emptyset$ is the empty set. It contains no strings.

Strings

- The length of a string $w$ (denoted by $|w|$) is the number of symbols in $w$.
- For integer $n \geq 0$, $\Sigma^n$ is set of all strings over $\Sigma$ of length $n$.
- $\Sigma^*$ is the set of all strings over $\Sigma$.
- $\epsilon$ is a string containing no symbols.
- $\emptyset$ is the empty set.

Definitions

- $\Sigma^*$ is the set of all strings of all lengths including empty string.
- $\epsilon$ is a string containing no symbols.
- $\emptyset$ is the empty set.

String operations

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

2 Overview of language complexity

Overview

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Languages</th>
<th>Production Rules</th>
<th>Automaton</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td>recursively enumerable</td>
<td>$\gamma \rightarrow \alpha$ (no constraints)</td>
<td>Turing machine</td>
<td>$L = { w</td>
</tr>
<tr>
<td>Type-1</td>
<td>context-sensitive</td>
<td>$\alpha A \beta \rightarrow \alpha \gamma \beta$</td>
<td>linear bounded nondeterministic Turing machine</td>
<td>$L = { a^n b^n c^n</td>
</tr>
<tr>
<td>Type-2</td>
<td>context-free</td>
<td>$A \rightarrow \alpha$</td>
<td>nondeterministic pushdown automata</td>
<td>$L = { a^n b^n</td>
</tr>
<tr>
<td>Type-3</td>
<td>regular</td>
<td>$A \rightarrow aB$</td>
<td>finite state machine</td>
<td>$L = { a^n</td>
</tr>
</tbody>
</table>

Meaning of symbols:
- $a$ - terminal
- $A$, $B$ - variables
- $\alpha$, $\beta$, $\gamma$ - strings in $\{a \cup A\}^*$ where $\alpha$, $\beta$ are maybe empty, $\gamma$ is never empty

Table borrowed from Wikipedia [https://en.wikipedia.org/wiki/Chomsky_hierarchy](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 denote a language. A regular expression \( r \) over an alphabet \( \Sigma \) is one of the following:

**Base cases:**
- \( \emptyset \) the language \( \emptyset \)
- \( \varepsilon \) denotes the language \( \{ \varepsilon \} \)
- \( a \) denote the language \( \{ a \} \)

**Inductive cases:** If \( r_1 \) and \( r_2 \) are regular expressions denoting languages \( L_1 \) and \( L_2 \) respectively (i.e., \( L(r_1) = L_1 \) and \( L(r_2) = L_2 \)) then,
- \( r_1 + r_2 \) denotes the language \( L_1 \cup L_2 \)
- \( r_1 \cdot r_2 \) denotes the language \( L_1 \cdot L_2 \)
- \( r_1^* \) denotes the language \( L_1^* \)

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

Non-deterministic 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 \( w \) if some accepting state is reached by \( N \) 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 non-deterministic finite automaton (NFA) \( N = (Q, \Sigma, s, A, \delta) \) is a five tuple where
- \( Q \) is a finite set whose elements are called states
- \( \Sigma \) is a finite set called the input alphabet
- \( \delta : Q \times \Sigma \cup \{ \varepsilon \} \rightarrow P(Q) \) is the transition function (where \( P(Q) \) is the power set of \( Q \))
- \( s \) and \( A \) are the same as in DFAs

**Example:**

\[
\begin{array}{c|ccc}
   & 0 & 1 & \varepsilon \\
\hline
0 & q_0 & q_1 & q_0 \\
1 & q_1 & q_0 & q_1 \\
\varepsilon & q_2 & q_2 & q_2 \\
\hline
s = q_0 \\
A = \{ q_1 \}
\end{array}
\]

For NFA \( N = (Q, \Sigma, \delta, s, A) \) and \( q \in Q \), the \( \varepsilon \)-reach\( (q) \) is the set of all states that \( q \) can reach using only \( \varepsilon \)-transitions.

Inductive definition of \( \delta^* : Q \times \Sigma^* \rightarrow P(Q) \):
- \( \delta^*(q, \varepsilon) = \varepsilon\)-reach\( (q) \)
- \( \delta^*(q, a) = \varepsilon\)-reach\( (q) \) for \( a \in \Sigma \)
- \( \delta^*(q, ax) = \varepsilon\)-reach\( (q) \) for \( a \in \Sigma, x \in \Sigma^* \)

**Example:**

\[
\begin{array}{c|ccc}
   & 0 & 1 & \varepsilon \\
\hline
0 & q_0 & q_1 & q_0 \\
1 & q_1 & q_0 & q_1 \\
\varepsilon & q_2 & q_2 & q_2 \\
\hline
s = q_0 \\
A = \{ q_1 \}
\end{array}
\]

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
- \( \Sigma \) is a finite set called the input alphabet
- \( \delta : Q \times \Sigma \rightarrow Q \) is the transition function
- \( s \in Q \) is the start state
- \( A \subseteq Q \) is the set of accepting/final states

**Example:**

\[
\begin{array}{c|cc}
   & 0 & 1 \\
\hline
0 & q_0 & q_1 \\
1 & q_1 & q_0 \\
\hline
s = q_0 \\
A = \{ q_1 \}
\end{array}
\]

Every string has a unique walk along a DFA. We define the extended transition function as \( \delta^* : Q \times \Sigma^* \rightarrow Q \) defined inductively as follows:
- \( \delta^*(q, \varepsilon) = q \) if \( w = \varepsilon \)
- \( \delta^*(q, ax) = \delta^*(\delta(q, a), x) \) if \( w = ax \)

Can create a larger DFA from multiple smaller DFAs. Suppose
- \( L(M_0) = \{ w \mid w \text{ has an even number of } 0 \} \) (picted above)
- \( L(M_1) = \{ w \mid w \text{ has an even number of } 1 \} \)

\[
\begin{array}{c|cc}
   & 0 & 1 \\
\hline
0 & (q_0, q_1) & (q_1, q_0) \\
1 & (q_1, q_0) & (q_0, q_1) \\
\hline
s = (q_0, q_1) \\
A = \{ q_0, q_1 \}
\end{array}
\]

Regular language equivalences

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

Fooling sets

Some languages are not regular (Ex. \( L = \{ 0^n1^n \mid n \geq 0 \} \)).

Two states \( p, q \in Q \) are distinguishable if there exists a string \( w \in \Sigma^* \) such that
- \( \delta^*(p, w) \in A \) and \( \delta^*(q, w) \notin A \).

Two states \( p, q \in Q \) are equivalent if for all strings \( w \in \Sigma^* \), we have that
- \( \delta^*(p, w) \in A \) if and only if \( \delta^*(q, w) \in A \).

For a language \( L \) over \( \Sigma \) 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.

Arden’s rule: If \( R = Q + RP \) then \( R = QP^* \).
### 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 \( A \rightarrow \alpha \) where \( A \in V \) and \( \alpha \) is a string in \( (V \cup T)^* \). Formally, \( P \subseteq V \times (V \cup T)^* \).
- \( S \in V \) is the start symbol

Example: \( L = \{ w w^R | w \in \{0, 1\}^* \} \) is described by \( G = (V, T, P, S) \) where:
- \( V = \{ S \} \)
- \( T = \{0, 1\} \)
- \( P = \{ S \rightarrow \epsilon, S \rightarrow 0S0, S \rightarrow 1S1 \} \) (abbreviation for \( S \rightarrow \epsilon, S \rightarrow 0S0, S \rightarrow 1S1 \))
- \( S = S \)

### Pushdown automata

A pushdown automaton is an NFA with a stack. The language \( L = \{ 0^n1^n | n \geq 0 \} \) 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
- \( \Sigma \) is a finite set called the input alphabet
- \( \Gamma \) is a finite set called the stack alphabet
- \( \delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow P(Q \times (\Gamma \cup \{\epsilon\})) \) 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 (input read), (stack pop) \rightarrow (stack push).

A CFG can be converted to a pushdown automaton.

### Context-free closure

Context-free languages are closed under union, concatenation, and Kleene star. They are not closed under intersection or complement.
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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log^2 n)$ if using MoM</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

We can divide and conquer multiplication like so:

$$bc = 10^n b_{LC} + 10^n/2 (b_{LCR} + b_{LC}) + b_{R}.\]$$

We can rewrite the equation as:

$$bc = (b_{LC}x + b_{CR})(x) + b_{LC}x + b_{CR} + b_{LC} + b_{CR} x + b_{R}.\]$$

Its running time is $O(n^{\log 2}) = O(n^{0.807}).$

Linear time selection

The median of medians (MoM) algorithms give a element that is larger than $\frac{3}{4}$'s and smaller than $\frac{1}{4}$'s of the array elements. This is used in the linear time selection algorithm to find element of rank $k$.

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

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

```python
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 + LIS_smaller(A[1..(n-1)], A[n]))
  return max
```

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:

```python
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)
```

### Definitions
- Problem instance of size $n$ is reduced to one or more instances of size $n-1$ or less.
- For termination, problem instances of small size are solved by some other method as base cases.

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

### 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 $f(n) = O(n^\alpha)$ for some $\alpha < \log r/c$, then $T(n) = O(n^\alpha \log n)$.

### Backtracking

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

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

### Simple recursion

- **Reduction:** solve one problem using the solution to another.
- **Recursion:** a special case of reduction - reduce problem to a smaller instance of itself (self-reduction).

### 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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log^2 n)$ if using MoM</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

We can divide and conquer multiplication like so:

$$bc = 10^n b_{LC} + 10^n/2 (b_{LCR} + b_{LC}) + b_{R}.\]$$

We can rewrite the equation as:

$$bc = (b_{LC}x + b_{CR})(x) + b_{LC}x + b_{CR} + b_{LC} + b_{CR} x + b_{R}.\]$$

Its running time is $O(n^{\log 2}) = O(n^{0.807}).$

### Linear time selection

The median of medians (MoM) algorithms give a element that is larger than $\frac{3}{4}$'s and smaller than $\frac{1}{4}$'s of the array elements. This is used in the linear time selection algorithm to find element of rank $k$.

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

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

```python
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 + LIS_smaller(A[1..(n-1)], A[n]))
  return max
```
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).
- **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 $u$, $\text{con}(u)$, 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 $\text{rch}(u)$ 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), v):
    toposort := empty list
    for v ∈ V:
        in(v) := |{w | w → v ∈ E}|
        while v ∈ V that has in(v) = 0:
            Add v to end of toposort
            Remove v from V
            for v in u → v in E:
                in(v) := 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 postvisit order.

Strongly connected components

- Given $G$, $w$ is strongly connected to $v$ if $v \in \text{rch}(w)$ and $w \in \text{rch}(v)$.
- A maximal group of vertices that are all strongly connected to one another is called a strong component.

**Pseudocode: Metagraph - linear time**

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

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 an unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

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

**Pseudocode: LIS - DP**

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

$$\text{Opt}(i, j) = \begin{cases} \alpha_{x_i, y_j} + \text{Opt}(i - 1, j - 1) & \text{if } x_i \neq y_j \\ \delta + \text{Opt}(i - 1, j) & \text{if } x_i = y_j \text{ and } i > 0 \\ \delta + \text{Opt}(i, j - 1) & \text{if } x_i = y_j \text{ and } i = 0 \end{cases}$$

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

**Pseudocode: Edit distance - DP**

```
EDIST(A[1..m], B[1..n])
    for i ← 1 to m do M[i, 0] = iδ
    for j ← 1 to n do M[0][j] = jδ
    for i = 1 to m do
        for j = 1 to n do
            M[i][j] = min \{\begin{align*}
                \text{COST}[A[i]][B[j]] + M[i-1][j-1], \\
                M[i-1][j] + δ, \\
                M[i][j-1] + δ
            \end{align*}\}
```

Graph algorithms
DFS and BFS

**Pseudocode: Explore (DFS/BFS)**

**Explore**($G$, $v$):
for $i ← 1$ to $|V|$
  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)

Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if an edge $(u, v)$ is:
- **Forward edge:** $\preceq u < \preceq v < \postceq u$
- **Backward edge:** $\preceq u < \postceq v < \preceq u$
- **Cross edge:** $\preceq u < \postceq v < \preceq v$

**Minimum Spanning Trees**

Some notes on minimum spanning trees:
- **Tree**: undirected graph in which any two vertices are connected by exactly one path.
- **Sub-graph $H$ of $G$**: spanning for $G$, if $H$ and $G$ have the same connected components.
- **Minimum spanning tree**: A spanning tree $T$ of minimum cost.
- **An edge $e = (u, v)$ in $T$ is safe if there is a unique minimum cost edge crossing $S$ (one end in $S$ and the other in $V \setminus S$).
- **An edge $e = (u, v)$ in $T$ is unsafe if there is some cycle $C$ such that $e$ is the unique maximum cost edge in $C$.
- **All edges are safe or unsafe.**

**Pseudocode: Borůvka’s algorithm:** $O(n \log n)$ using Union-Find structure.

$T$ is $\emptyset$ (i.e., $T$ will store edges of a MST$^*$)
while $T$ is not spanning do
  for each connected component $S$ of $T$ do
    add to $T$ the cheapest edge between $S$ and $V \setminus S$.
  end for
end while
return the set $T$

**Pseudocode: Kruskal’s algorithm:** $O(m \log n)$ using Union-Find structure.

Sort edges in $E$ based on cost
$T$ is empty (i.e., $T$ will store edges of a MST$^*$)
while $E$ is not empty do
  pick $e = (u, v) \in E$ of minimum cost
  if $u$ and $v$ belong to different sets
    add $e$ to $T$
  end if
  merge the sets containing $u$ and $v$
end while
return the set $T$

**Pseudocode: Prim’s algorithm:** $O(|E| \log |V|)$ using Priority Queue.$^*$

$T ← \emptyset$, $S ← \emptyset$, $\alpha ← 1$
$\forall u \in V \setminus S$: $d(u) ← \infty$, $p(u) ← \emptyset$
$d(s) ← 0$
while $S \neq V$ do
  $v = \arg \min_{u \in V \setminus S} d(u)$
  $T ← T \cup \{vp(v)\}$
  $S ← S \cup \{v\}$
  for each $u \in Adj(v)$ do
    $d(u) ← \min\{d(u), d(v) + c(uv)\}$
    $p(u) ← v$
  end for
end while

Shortest Paths

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

**Pseudocode: Dijkstra**

for $v \in V$ do
  $d(v) ← \infty$
  $X ← S \cup \{v\}$
  $d(s, s) ← 0$
for $i ← 1$ to $n$ do
  $v ← \arg \min_{u \in V \setminus X} d(u)$
  $X ← X \cup \{v\}$
  for each $(u, v) \in Adj(v)$ do
    $d(u) ← \min\{d(u), d(v) + f(u, v)\}$
end for
return $d$

**Running time:** $O(|V| + |E| \log |E|)$ (if using a Fibonacci heap as the priority queue)

**Bellman-Ford algorithm**:
Find minimum distance from vertex $s$ to all other vertices in graphs with negative edge weights.

**Pseudocode: Bellman-Ford**

for each $v \in V$ do
  $d(v) ← \infty$
  $d(s) ← 0$
for $k ← 1$ to $n - 1$ do
  for each $(u, v) \in E$ do
    $d(v) ← \min\{d(v), d(u) + f(u, v)\}$
end for
end while

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

**Floyd-Warshall algorithm**:
Find minimum distance from every vertex to every vertex in a graph with negative cycles.
It is a DP algorithm with the following recurrence:

$d(v, k) = \begin{cases} 0 & \text{if } v = s \text{ and } k = 0 \\ \infty & \text{if } v \neq s \text{ and } k = 0 \\ \min\{d(u, k - 1) + f(u, v)\} & \text{otherwise} \end{cases}$

**Running time**: $O(|V|^3)$

**Metagraphs ($G(V, E)$)**:

**Pseudocode: Floyd-Warshall**

for each $i \in V$ do
  for each $j \in V$ do
    $d(i, j, k) ← \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (i, j) \not\in E \text{ and } k = 0 \\ \min\{d(i, j, k - 1) + d(k, j, k - 1)\} & \text{otherwise} \end{cases}$
end for
end for

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

**Pseudocode: Floyd-Warshall**

for each $v \in V$ do
  if $d(i, i, n - 1) < 0$ then
    return “Negative cycle in $G$”
end for

**Running time**: $\Theta(n^3)$
Complexity Classes

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

Undecidable

NP-Hard

NPC

P

PSPACE

EXP

Decidable (Recursive)

Semi-Decidable (recursively-enumerable, recognizable, Turing-acceptable/recognizable, partially-decidable)

Context-Sensitive

Context-Free

Regular

 reductions

Algorithms

Complexity Classes

Computational Complexity Classes

Turing-recognizable

PSPACE

co-NP

P

NP

Turing-acceptable/recognizable, partially-decidable)

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

Sample undecidable problems

- ACCEPTInput: $A_{TM} = \{ (M, w) \mid M$ is a TM and $M$ accepts $w \}$
- HALTInput: $Hal_{TM} = \{ (M, w) \mid M$ is a TM and $M$ halts on input $w \}$
- HALTInput: $Hal_{TM} = \{ (M) \mid M$ is a TM and halts on blank input $\}$
- EMPTINESS: $E_{TM} = \{ (M) \mid M$ is a TM and $L(M) = \emptyset \}$
- EQUALITY: $EQ_{TM} = \{ (M_A, M_B) \mid M_A$ and $M_B$ are TMs and $L(M_A) = L(M_B) \}$