Week 1: Recap, Binary Radix Sort, Tries

Data Structures:

-

Sorting algorithms:

- Naïve – O(n^2) in the worst/average case
  - Selection sort, insertion sort, bubble sort
- Clever – O(n logn) in the worst/average case
  - Merge sort, Heapsort
- Fastest in practice – O(n log n) on average, no better than O(n^2) in the worst case unless a clever variant is used
  - Quicksort

Comparison-based sorting

- **No sorting algorithm based on pairwise comparison of values can be better than O(n log n)**
- Proof in Vid 2

Radix sort

- O(n)
- Has to exploit the structure of the items being sorted, so may be less versatile
- In practice, faster than O(n log n) only for very large n
- Algorithm:
  1. Assume items to sort can be treated as bit-sequences of length m
  2. Let b be a chosen factor of m
     a. b and m are constants for any particular instance
  3. (Each item has bit positions labelled 0, 1, m-1 (right to left))
  4. m/b iterations
     a. In each iteration, the items are distributed into 2^b buckets (lists)
        i. The buckets are labelled $0,1,\dots,2^b-1$
     b. During the i^{th} iteration, an item is placed in the bucket corresponding to the integer represented by the bits in positions $b \times i - 1, \dots, b \times (i-1)$
     c. At the end of an iteration, the buckets are concatenated to give a new sequence for the next iteration
- Pseudocode:

- ```java
  // assume we have the following method which returns the value
  // represented by the b bits of x when starting at position pos
  private int bits(Item x, int b, int pos)

  // suppose that:
  //     a is the sequence to be sorted
  //     m is the number of bits in each item of the sequence a
  //     b is the 'block length' of radix sort

  int numIterations = m/b; // number of iterations required for sorting
  int numBuckets = (int) Math.pow(2, b); // number of buckets

  // represent sequence a to be sorted as an ArrayList of Items
  ArrayList<Item> a = new ArrayList<Item>();

  // represent the buckets as an array of ArrayLists
  ArrayList<Item>[] buckets = new ArrayList[numBuckets];
  for (int i=0; i<numBuckets; i++) buckets[i] = new ArrayList<Item>();
  ```
- ```java
  for (int i=1; i<=numIterations; i++){

     // clear the buckets
     for (int j=0; j<numBuckets; j++) buckets[j].clear();

     // distribute the items (in order from the sequence a)
     for (Item x : a){
       // find the value of the b bits starting from position (i-1)*b in x
       int k = bits(x, b, (i-1)*b); // find the correct bucket for item x
       buckets[k].add(x); // add item to this bucket
     }

     a.clear(); // clear the sequence

     // concatenate the buckets (in sequence) to form the new sequence
     for (j=0; j<numBuckets; j++) a.addAll(buckets[j]);

  }
  ```

- Correctness
  - [Proof in Vid 3]
- Complexity
  - O(n)
    - Number of buckets - 2^b
    - Number of iterations – m/b
    - During each iteration:
      - Bucket allocation – O(n)
      - Bucket concatenation – O(2^b)
    - Overall: $O\left(m/b \cdot \left(n + 2^b\right)\right)$
  - **Time-space trade-off**
    - The larger the value of b, the smaller the multiplicative constant (m/b) in the complexity function and so the faster the algorithm will become
    - However, an array of size 2^b is required for the buckets; therefore, increasing b will increase the space requirements

Tries (re**trie**val)

- Tries are to binary trees (comparison-based) as Radix sort is to comparison-based sorting
- Basics:
  - Stored items have a key value that is interpreted as a sequence of bits/character
  - Multiway branch at each node, where each branch has an associated symbol, and no 2 siblings have the same symbol
  - The branch taken at level i during a search is determined by the i^{th} element of the key value (i^{th} bit, i^{th} char)

    - Tracing a path from the root to a node spells out the key value of the item
- Search pseudocode:
    -
```
// searching for a word w in a trie t
Node n = root of t; // current node (start at root)
int i = 0; // current position in word w (start at beginning)

while (true) {
  if (n has a child c labelled w.charAt(i)) {
    // can match the character of word in the current position
    if (i == w.length()-1) { // end of word
      if (c is an 'intermediate' node) return "absent";
      else return "present";
    }
    else { // not at end of word
      n = c; // move to child node
      i++; // move to next character of word
    }
  }
  else return "absent"; // cannot match current character
}
```
- Insert pseudocode:
    -
```
// inserting a word w in a trie t
Node n = root of t; // current node (start at root)

for (int i=0; i < w.length(); i++){ // go through chars of word
  if (n has no child c labelled w.charAt(i)){
    // need to add new node
    create such a child c;
    mark c as intermediate;
  }
  n = c; // move to child node
}
mark n as representing a word;
```
- Complexity of operations
    - Almost independent of no. of items
    - Essentially linear in the string length
- Implementation
    - Array (of pointers to represent the children of each node)
    - Linked lists (to represent the children of each node)
    - Time/space trade-off

Week 2: Strings and Text Algorithms

Text compression

- A special case of **data compression**
    - Saves disk space and transmission time
- Must be **lossless** (original must be recoverable without error)
- Some other forms of compression can be lossy (pictures, sound, etc)
- Examples:
    - compress, gzip in Uniz, ZIP utilities for Windows
    - 2 main approaches: statistical and dictionary
- **Compression ratio**: x/y
    - x – size of compressed file, y – size of original file
    - measured in B, kB, MB, …
    - Compressing 10MB to 2 would yield 2/10=0.2
- **Percentage space saved**: (1 – “compression ratio”) * 100%
    - Space saved expressed as a percentage of the original file size

    - 10 to 2MB -> 80%
- Space savings in the range 40-60% are typical
    - The higher the saving, the better
- Huffman encoding
    - Classical **statistical** method
        - Now mostly superseded in practice by more effective dictionary methods
        - Fixed (ASCII) code replaced by **variable** length code for each char
        - Every char is represented by a unique codeword (bit string)
        - Frequently occurring chars are represented by shorter codewords
    - The code has the **prefix** property
        - No codeword is a prefix of another (gives **unambiguous** decompression)
    - Based on a **Huffman tree** (a proper binary tree)
        - Each char is represented by a leaf node
        - Codeword for a char is given by the path from the root to the appropriate leaf (left=0, right=1)
        - Prefix property follows from this
        - Construction:
            1. Add leaves of Huffman tree
                a. Characters with their frequencies label the leaf nodes
            2. While there is >1 parentless node:
                a. Add new parent to nodes of smallest weight
                b. Weight of new node equals sum of the weights of the child nodes

```
// set up the leaf nodes
for (each distinct character c occurring in the text){
  make a new parentless node n;
  int f = frequency count for c;
  n.setWeight(f); // weight equals the frequency
  n.setCharacter(c); // set character value
  // leaf so no children
  n.setLeftChild(null);
  n.setRightChild(null);
}
// construct the branch nodes and links
while (no. of parentless nodes > 1){
  make a new parentless node z; // new node
  x, y = 2 parentless nodes of minimum weight; // its children
  z.setLeftChild(x); // set x to be the left child of new node
  z.setRightChild(y); // set y to be the right child of new node
  int w = x.getWeight()+y.getWeight(); // calculate weight of node
  z.setWeight(w); // set the weight of the new node
}
```

    - Optimality
        - Weighted path length (WPL) of a tree T
            - Sum of weight * distance from root, where sum is over all leaf nodes
        - Huffman tree has minimum WPL over all binary trees with the given leaf weights
            - Huffman tree need not be unique (e.g., nodes > 2 with min weight)
            - However, all Huffman trees for a given set of frequencies have same WPL
            - WPL is the number of bits in compressed file
                - bits = sum over chars (frequency of chars * code length of char)
            - A Huffman tree **minimises** this number

            - Hence, Huffman coding is **optimal** for all possible codes built this way
    - Algorithmic requirements (n – text length; m – distinct chars in text)
        - Building the tree – **O(n + m log m)**
            - Based on using a heap for the weights
            - O(n) for finding frequencies and then O(m log m) to build the tree
            - Since m is essentially a constant, it is really **O(n)**
        - Compression uses a code table (an array of codes, indexed by char)
            - O(m log m) to build the table
                - m chars, so m paths of length O(log m)
                - The Huffman tree is also a binary tree, so height is O(log m)
            - O(n) to compress: n chars in the text, so n lookups in array – O(1)
            - Overall: **O(m log m) + O(n)**
        - Decompression uses the tree directly (repeatedly trace paths in tree)
            - **O(n log m)** since n is size of text, hence, n paths of length O(log m)
        - Problem – some representation of the Huffman tree must be stored with the compressed file (otherwise, decompression would be impossible)
            - Alternatives:
                - Fixed set of frequencies based on typical values for text
                    - Reduces compression ratio
                - Adaptive Huffman coding: the same tree is built and adapted by compressor and by the decompressor as chars are encoded/decoded
                    - Slows down compression and decompression (but not by much if clever)
- LZW compression (Lempel, Ziv, and Welch)
    - Popular dictionary-based method
    - The dictionary is a collection of strings
        - Each with a **codeword** that represents it
        - The codeword is a bit string
        - It can be interpreted as a non-negative integer
    - Whenever a codeword is outputted during compression, the **bit string** is written of the compressed file
        - Using a number of bits determined by the **current codeword length**
        - At any point all bit strings are the same length
    - The dictionary is built **dynamically** during compression
        -
```
set current text position i to 0;
initialise codeword length k (say to 8);
initialise the dictionary d;

while (the text t is not exhausted) {

  identify the longest string s, starting at position i of text t
  that is represented in the dictionary d;
  // there is such string, as all strings of length 1 are in d

  output codeword for the string s; // using k bits

  // move to the next position in the text
  i += s.length(); // move forward by the length of string just encoded
  c = character at position i in t; // character in next position

  add string s+c to dictionary d, paired with next available codeword;
  // may have to increment the codeword length k to make this possible
}
```

- (and also during decompression)

```
initialise codeword length k;
initialise the dictionary;

read the first codeword x from the compressed file f; // i.e. read k bits
String s = d.lookUp(x); // look up codeword in dictionary
output s; // output decompressed string

while (f is not exhausted){

  String oldS = s.clone(); // copy last string decompressed

  if (d is full) k++; // dictionary full so increase the code word length

  get next codeword x from f;  // i.e. read k bits
  s = d.lookUp(x); // look up codeword in dictionary
  output s; // output decompressed string

  String newS = oldS + s.charAt(0); // string to add to dictionary

}
```

- Initially, dictionary contains all possible strings of length 1
- Throughout execution, the dictionary is **closed under prefixes**
  - i.e., if the string s is represented in the dictionary, so is every prefix of s
- => a **trie** is an ideal representation of the dictionary
  - **every** node is the trie represents a 'word' in the dictionary
  - a trie is effective and efficient ??? [in live lec]

- During (de)compression there is a current **codeword length k**. This value is dynamic
  - Codewords are bit strings of length k
    - Clear how many bits to read when decompressing
  - For a given k, there are exactly $\mathbf{2^k}$ distinct codewords available
    - i.e., all possible bit strings of length k
    - This limits the size of the dictionary
    - However, the codeword length can be incremented when necessary
    - thus doubling the number of available codewords
    - and **all** codewords change (bit strings now of length k+1)
  - Initial value of k should be large enough to encode all strings of length 1
- Variants:
  - Constant codeword length
    - Fix the codeword length for all time
    - The dictionary has fixed capacity: when full, just stop adding to it
  - **Dynamic codeword length**
    - Start with shortest reasonable codeword length, say, 8 for normal text
    - When dictionary becomes full,
      - add 1 to current codeword length (2x no. of codewords)
      - does not affect the sequence of codewords already output
    - May specify a maximum codeword length, as increasing the size indefinitely may become counter-productive
  - LRU (Least Recently Used)
    - When dictionary becomes full and codeword length maximal
    - Current string replaces LRU string in dictionary
- Decompression – Special Case
  - Possible to encounter a codeword that is not (yet) in the dictionary

            - because decompression is ‘out of phase’ with compression
            - Possible to deduce what string it must represent
        - Solution: if (lookup fails) s = oldS + oldS.charAt(0);
    - Complexity
        - Complexity of compression and decompression both **O(n)**
            - For a text of length n (if suitably implemented)
            - Algorithms essentially involve just 1 pass through the text
            - Need to “search” for longest strings in the dictionary and then add new strings to the dictionary
                - Efficiently performing this search and adding a string is part of the AE

String comparison

- Notation:
    - s=$s_0 s_1 \ldots s_{m-1}$
        - m – length of string
        - s[i] is the (i+1)th element of the string, i.e., s_i
        - s[i…j] – substring from i to j
    - Prefixes and suffixes
        - j-th prefix – the first j characters of s denoted s[0…j-1]
            - 0^{th} prefix – empty string
        - j-th suffix – the last j characters of s denoted s[m-j…m-1]
            - 0^{th} suffix – empty string
- Fundamental question: how similar (different) are 2 strings?
    - Applications include:
        - biology (DNA and protein sequences)
        - file comparison (diff in Unix)
        - spelling correction, speech recognition, …
- **Basic operations** for transforming
    - **insert** a char
    - **delete** a char
    - **substitute** a char by another
- String distance
    - **Distance** between s and t – the smallest number of basic operations needed to transform s to t
    - String comparison algorithms use **dynamic programming**
        - Problem solved by building up solutions to sub-problems of ever-increasing size
        - Often called the **tabular method** (builds up a **table** of relevant values)
        - Eventually, one of the values in the table gives the required answer
    - **Dynamic programming**
        - i-th prefix of string s is the first i characters of s
            - Let **d(i,j)** be the distance between i-th prefix of s and j-th prefix of t
            - Distance between s and t is then d(m,n) (since lengths of s and t are m and n)
        - Recurrence relation:

- $$d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{ d(i,j-1), d(i-1,j), d(i-1,j-1) \} & \text{otherwise} \end{cases}$$

  no operations are required
  the distance is given by that between the $i-1^{th}$ and $j-1^{th}$ prefixes of s and t

    - Second line:
        - '1' will perform 1 operation (insertion/deletion/substitution)
        - d(i,j-1) – insertion
        - d(i-1,k) – deletion
        - d(i-1,j-1) – substitution

- Algorithm comes immediately from the formula
- Time and space complexity both **O(mn)**
    - A consequence of the size of the table
    - Space complexity can easily be reduced to **O(m+n)** (by keeping the most recent entry in each column of the table)
- **Traceback phase** used to construct an **optimal alignment**
    - Trace a path in the table from bottom right to top left
    - Draw an arrow from an entry to the entry that led to its value
    - Can be done using only O(m+n) space with Hirschberg's algorithm
- Interpretation
    - Vertical step – deletion
    - Horizontal step – insertion
    - Diagonal step – match/substitution
        - A match if the distance doesn't change, substitution otherwise

Week 3: String/Pattern Search

Def:

- Searching a (long) text for a (short) string/pattern
    - Many applications, including:
        - info retrieval
        - text editing
        - computational biology
    - Many variants, such as exact/approximate matches
        - 1st occurrence or all occurrences
        - 1 text and many string/patterns
        - many texts and 1 string/pattern
    - Abstract view
        - given a text t (of length n) and a string/pattern s (of length m), find the position of the first occurrence (if it exists) of s in t
            - usually n is large and m is small

- Algorithms:
  - Brute force (exhaustive search)
    - Tests all possible positions
    - Wordy:
      - Set the current starting position in text as 0
      - Compare text and string chars left-to-right until the entire string is matched/character mismatches
      - If mismatch:
        - Advance the starting position by 1 and repeat
      - Continue until a match or text is exhausted
    - Uses char arrays in Java (not strings)

```java
/** return smallest k such that s occurs in t starting at position k */
public int bruteForce (char[] s, char[] t){
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int sp = 0;  // starting position in text t
   int i = 0;   // curr position in text
   int j = 0;   // curr position in string/pattern s
   while (sp <= n-m && j < m) { // not reached end of text/string
      if (t[i] == s[j]){ // chars match
         i++;      // move on in text
         j++;      // move on in string/pattern
      } else {     // a mismatch
         j = 0;    // start again in string
         sp++;     // advance starting position
         i = sp;   // back up in text to new starting position
      }
   }
   if (j == m) return sp; // occurrence found (reached end of string)
   else return -1; // no occurrence (reached end of text)
}
```

    -
    - Complexity
      - Worst case is no better than O(mn)
      - Typically, the number of comparisons from each point will be small
        - Often just 1 comparison for mismatch
        - => O(n) on average
  - KMP
    - Knuth-Morris-Pratt
    - O(m+n) in the worst case
    - **on-line** – removes the need to back-up in the text
      - Involves pre-processing the string to build a border table
      - **Border table**: an array b with entry b[j] for each position j of the string
    - If mismatch at position j in string/pattern:
      - remain on the current text char (do not back-up)
      - The border table shows which string char should next be compares with the current text char
    - Border – a substring that is both a prefix and a suffix (but not the whole string)
      - Some strings don't have borders, then the empty string (length 0) is the longest
    - String search – Border table
      - Border table b of the string pattern

        - b[j] = length of the longest border of s[0…j-1]

```java
/** return smallest k such that s occurs from position k in t or -1 if no k exists */
public int kmp(char[] t, char[] s) {
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int i = 0; // current position in text
   int j = 0; // current position in string s
   int [] b = new int[m]; // create border table
   setUp(b); // set up the border table
   while (i <= n) { // not reached end of text
      if (t[i] == s[j]){ // if positions match
         i++; // move on in text
         j++; // move on in string
         if (j = m) return i - j; // reached end of string so a match
      } else { // mismatch adjust current position in string using the border table
          if (b[j] > 0) // there is a common prefix/suffix
              j = b[j]; // change position in string (position in text unchanged)
          else { // no common prefix/suffix
              if (j = 0) i++; // move forward one position in text if not advanced
              else j = 0; // else start from beginning of the string
          }
      }
   }
   return -1; // no occurrence
}
```

      - Search – O(n) in worst case
      - Border table creation
        - Naïve – O(m^3)
        - More efficient: use KMP algorithm => O(m)
    - Boyer-Moore
      - Almost always **faster** than brute force/KMP
      - Many apps
      - Properties:
        - Typically, many text characters are skipped without even being checked
        - String/pattern is scanned **right-to-left**
        - Text characters involved in a mismatch are **used to decide** next comparison
        - Involves pre-processing the string to record the position of the **last** occurrence of each character c in the alphabet
          - The alphabet must be fixed in advance of the search
        - Last occurrence position of character c in the string s (-1 if not present)
      - Simplified version – Boyer-Moore-Horspool
      - Complexity
        - Worst case – O(mn)
          - Using good suffix rule, linear (O(m+n))

Week 4: Graphs and Graph Algorithms

Recap of AF2:

- Undirected graph G = (V,E)
  - V – finite set of vertices (vertex set)
  - E – set of edges, each edge is a subset of V of size 2 (edge set)
  - Pictorially

        - V – point
        - Edge – line connecting points
        - Representations of same graph can be different
- Properties
    - Vertices are **adjacent** if there is an edge connecting them
        - **Non-adjacent** – not
    - Vertex is **incident** to edge if the edge connects it
    - Path (length = number of edges)
        - Cycle – path that ends where it starts
    - **Degree** – number of edges a vertex is incident to
    - Connected graph if every pair of vertices is joined by a path
        - Non-connected – 2+ connected components
    - **Tree** – connected and acyclic (no cycles) graph
        - Tree with n vertices has n-1 edge
    - **Forest** – acyclic and components are tree
    - **Complete** (a **clique**) – every pair of vertices is joined by an edge
    - **Bipartite** – vertices are in 2 disjoint sets U & W and every edge joins a vertex in U to a vertex in W
- Directed graph (digraph) D = (V, E)
    - each edge is an ordered pair (x, y) of vertices
    - "adjacent to" and "adjacent from"
    - Pictorially
        - Edges are drawn as directed lines/arrows
    - Paths and cycles must follow edge directions

Graph representations

- Undirected graphs:
    - Adjacency matrix
        - 1 row and column for each vertex
        - row i, column j contains a 1 if i-th and j-th vertices are adjacent, 0 otherwise
    - Adjacency lists
        - 1 list for each vertex
        - list i contains an entry for j if the vertices i and j are adjacent
- Directed graphs
    - Adjacency matrix
        - 1 row, col for each vertex
        - row i, col j contains 1 if there is an edge from i to j, 0 otherwise
    - Adjacency lists
        - 1 list for each vertex
        - list for vertex i contains vertex j if there is an edge from i to j

Implementation – Adjacency lists

- Define classes for:
    - entries of adjacency lists

- - ```java
    /** class to represent an entry in the adjacency list of a vertex
    in a graph */
    public class AdjListNode {

      private int vertexIndex; // the vertex index of the entry

      // possibly other fields, for example representing properties
      // of the edge such as weight, capacity, …

      /** creates a new entry for vertex indexed i */
      public AdjListNode(int i){
        vertexIndex = i;
      }
      public int getVertexIndex(){ // gets the vertex index of the entry
        return vertexIndex;
      }
      public void setVertexIndex(int i){ // sets vertex index to i
        vertexIndex = i;
      }
    }
    ```
  - vertices (includes linked list representing its adjacency list)
    - ```java
      import java.util.LinkedList; // we require the linked list class

      /** class to represent a vertex in a graph */
      public class Vertex {

        private int index; // the index of this vertex
        private LinkedList<AdjListNode> adjList; // the adjacency list of vertex

        // possibly other fields, e.g. representing data stored at the node

        /** create a new instance of vertex with index i */
        public Vertex(int i) {
          index = i; // set index
          adjList = new LinkedList<AdjListNode>();// create empty adjacency list
        }

         /** return the index of the vertex */
        public int getIndex(){
          return index;
        }
      ```
    - ```java
      // class Vertex continued

        /** set the index of the vertex */
        public void setIndex(int i){
         index = i;
        }
        /** return the  adjacency list of the vertex */
        public LinkedList<AdjListNode> getAdjList(){
         return adjList;
        }
        /** add vertex with index j to the adjacency list */
        public void addToAdjList(int j){
          adjList.addLast(new AdjListNode(j));
        }
        /** return the degree of the vertex */
        public int vertexDegree(){
          return adjList.size();
        }
      }
      ```
  - graphs (includes size of graph and an array of vertices)
    - array allows for efficient access of “index” of vertex
    - ```java
      import java.util.LinkedList; // again require the linked list class
      // (to add graph algorithms we will need to access adjacency lists)
      /** class to represent a graph */
      public class Graph {

        private Vertex[] vertices; // the vertices
        private int numVertices = 0; // number of vertices
        // possibly other fields representing properties of the graph

        /** Create a Graph with n vertices indexed 0,...,n-1  */
        public Graph(int n) {
          numVertices = n;
          vertices = new Vertex[n];
          for (int i = 0; i < n; i++) vertices[i] = new Vertex(i);
        }
        /** returns number of vertices in the graph */
        public int size(){
          return numVertices;
        }
      }
      ```

Graph search and traversal algorithms

- A systematic way to explore a graph (when starting from some vertex)
- Applications: web crawler with HTML
- Search/traversal visits all vertices by travelling along edges
  - Traversal is **efficient** if it explores graph in O(|V|+|E|) time
- **Depth-First Search** (DFS)
  - From starting index,
    - Follow a path of unvisited vertices **until path can be extended no further**
    - Then backtrack along the path until an unvisited vertex can be reached
    - Continue until cannot findy any unvisited vertices
  - Edges traversed form a spanning tree/forest

- Depth-first spanning tree (forest)
- Spanning tree of a graph is a tree composed of all vertices and some (or all) of the edges of the graph

- Implementation
  - Add to vertex class

```java
private boolean visited; // has vertex been visited in a traversal?

private int pred; // index of the predecessor vertex in a traversal

public boolean getVisited(){
  return visited;
}
public void setVisited(boolean b){
  visited = b;
}
public int getPred(){
  return pred;
}
public void setPred(int i){
  pred = i;
}
```

  - Add to graph class

```java
/** visit vertex v, with predecessor index p, during a dfs */
private void visit(Vertex v, int p){
  v.setVisited(true); // update as now visited
  v.setPred(p);  // set predecessor (indicates edge used to find vertex)
  LinkedList<AdjListNode> L = v.getAdjList(); // get adjacency list

  for (AdjListNode node : L){ // go through all adjacent vertices
    int i = node.getIndex(); // find index of current vertex in list
    if (!vertices[i].getVisited()) // if vertex has not been visited
       visit(vertices[i], v.getIndex()); // continue dfs search from it
       // setting the predecessor vertex index to the index of v
  }
}
/** carry out a depth first search/traversal of the graph */
public void dfs(){
  for (Vertex v : vertices) v.setVisited(false); // initialise
  for (Vertex v : vertices) if (!v.getVisited()) visit(v,-1);
  // if vertex is not yet visited, then start dfs on vertex
  // -1 is used to indicate v was not found through an edge of the graph
}
```

- Complexity
  - Overall O(n+m)
    - n – number of vertices
    - m – number of edges
  - Can be adapted to the adjacency matrix representation
    - Then it's O(n^2) to look at every entry
  - Some apps:
    - Determine if a given graph is connected
    - Identify the connected components of a graph
    - Determine if a given graph contains a cycle
    - Determine if a given graph is bipartite

- **Breadth-First Search (BFS)**
  - Search fans out as widely as possible at each vertex
    - From the current vertex, visit all the adjacent vertices (**processing** the current vertex)
    - Vertices are processed in the order in which they are visited (visited vertices added to/removed from a queue)
    - Continue until all vertices in current component have been processed
    - Repeat for other components
  - Edges traversed form a spanning tree (forest)
    - Breadth-first spanning tree (forest)
  - Implementation
    - [AE]
  - Complexity

        - Overall O(n+m)
            - n – number of vertices
            - m – edges
            - Each vertex is visited and queued exactly once
        - Example application: Finding **distance** between 2 vertices (v and w)
            - Assign distance to v to be 0
            - Carry out BFS from v
            - When visiting a new vertex, assign distance++
            - Stop when w is reached

Weighted graphs

- Each edge has weight by wt(e)>0
    - Graph undirected or directed
    - Represents lots of things (distance, cost)
    - If an edge is not part of the graph, its weight is infinity
- Representation
    - Adjacency matrix becomes weight matrix
    - Adjacency lists include weight in node
- Shortest paths
    - Length of a span – sum of weights of edges
- Dijkstra's algorithm
    - in 20 minutes, 26 y/o
    - Shortest path between 1 vertex (u) and all others
        - Maintains a set S containing all vertices for which shortest path with u is currently known
        - S initially contains only u
    - Each vertex v has a label d(v) indicating length of shortest path from u to v passing only through vertices in S
        - If no path exists, d(v) = infinity
        - If v is in S, then d(v) is length of shortest path between u and v
    - Invariant of the algorithm: if v is in S and w is not, then the length of the shortest path between u and w is at least that between u and v
        - At each step we add to S the vertex v not in S such that d(v) is minimum
        - After adding v to S, carry out edge relaxation operations (update length d(w) for all vertices w still not in S)
    - Edge relaxation
        - Suppose v and w are not in S, then we know
            - Shortest path between u and v passing only through S equals d(v)
            - shortest path between u and w passing only through S equals d(w)
        - Suppose v added to S and edge e = {v, w} has weight wt(e)
        - Calculate shortest path between u and w passing only though S union {v}
        - Shortest path is either:
            - original path through S of length d(w), or
            - path combining e and shortest path between v and u which has length wt(e) + d(v)
        - d(w) = min(d(w), d(v) + wt(e))
    - Implementation

- ```
  // S is set of vertices for which shortest path with u is known
  // d(w) represents length of a shortest path between u and w
  // passing only through vertices of S

  S = {u}; // initialise S
  for (each vertex w) d(w) = wt(u,w); // initialise lengths

  while (S != V){ // still vertices to add to S
    find v not in S with d(v) minimum;
    add v to S;
    for (each w not in S and adjacent to v) // perform relaxation
      d(w) = min{ d(w) , d(v)+wt(v,w) };
  }
  ```
- Complexity (n vertices, m edges)
  - Using unordered array for lengths:
    - O(n) to initialise
    - O(n^2) to find minimum
      - Each time it takes O(n) and there are n-1 to find
    - O(m) for relaxation
      - Each edge is considered once and updating length takes O(1)
    - Overall: **O(n^2)** (at most n(n-1) edges)
  - Using a heap for lengths
    - O(n) to initialise lengths and create heap
    - Finding minimum – O(n log n)
      - each time O(log n), n-1 to find
    - Relaxation – O(m log n)
      - Each edge is considered once, and updating lengths is O(log n)
    - Overall: **O(m log n)** (more edges than vertices)
      - A graph with n vertices has O(n^2) edges
- Correctness
  - [example in live lecture]

Week 5: Graph Algorithms

Spanning tree:

- subgraph (subset of edges) which is both a tree and 'spans' every vertex
- a **spanning tree** is obtained from a connected graph by deleting edges
- the **weight** of a spanning tree – sum of the weights of its edges
- Problem – **minimum weight spanning tree** from a weighted connected undirected graph
  - Represents cheapest way of interconnecting vertices
- Apps:
  - design of networks for computers, telecommunications, transportation, gas, electricity
  - clustering, approximating the travelling salesman problem

Minimum weight spanning tree problem

- An example of a problem in **combinatorial optimisation**
  - find best way of doing something among a (large) number of candidates
  ```

  - can always be solved, at least in theory, by exhaustive search
    - may be infeasible in practice
    - typically an exponential-time algorithm
    - e.g., K_n (clique of size n) has n^{n-2} spanning trees (Cayley's formula)
      - graph is a clique if every pair vertices is joined by an edge
  - Prim-Jarnik minimum spanning tree algorithm
    - example of a greedy algorithm
    - makes a sequence of decisions based on local optimality
    - ends up with the globally optimal solution
  - For many problems, greedy algorithms do not yield optimal solution

Prim-Jarnik algorithm

- Pseudocode:

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
while (number of ntv > 0){
    find edge e = {p,q} of graph such that
        p is a tv;
        q is an ntv;
        wt(e) is minimised over such edges;
    adjoin edge e to the (spanning) tree;
    make q a tv;
}
```

- Analysis:
  - initialisation O(n)
  - outer loop n-1
  - inner loop O(n^2) (all edges from a tree-vertex to a non-tree-vertex)
  - updating tree O(1)
  - Overall: $O(n^3)$
- Correctness:
  - [proof fnot part of the exam]
  - Yes
- Dijkstra's Refinement:
  - Introduce an attribute bestTV for each non-tree vertex q
    - bestTV is set to the tv p for which wt({p,q}) is minimised
  - Pseudocode:

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
for (each ntv s) set s.bestTV = r; // r is the only tv

while (number of ntv > 0){
  find ntv q for which wt({q, q.bestTV}) is minimal;
  adjoin {q, q.bestTV} to the tree;
  make q a tv;

  for (each ntv s) update s.bestTV;
  // update bestTV as tree vertices have changed
}
```

  - Analysis:
    - initialisation O(n)
    - while loop n-1
    - first part in while O(n)
      - O(n) to find minimal ntv and O(1) to ???
    - second part O(n)
    - Overall: $O(n^2)$

Topological ordering

- **Directed Acyclic Graph (DAG)** – directed graph with no cycles
- **Topological order** on a DAG is a labelling of the vertices 1,…,n such that $(u, v) \in E$ implies label(u) < label(v)
    - Many apps: scheduling, PERT networks, deadlock detection
        - Scheduling and PERT networks:
            - can model a project with a DAG
                - vertices are tasks/activities, edges indicate dependencies
            - if topological order, can find longest path or longest weighted path
            - can then determine which activities are "critical" (i.e., on longest paths)
- A directed graph D has a topological order if and only if it is a DAG
- **Source** – vertex of in-degree 0 and a sink has out-degree 0
- **DAG has at least 1 source and at least 1 sink**
    - forms the basis of a topological ordering algorithm
    - if there is no source/sink, can build a cycle => not acyclic
        - if no source or sink, can always keep adding vertices to the start/end of a path respectively
        - eventually must add the same vertex twice to the path as there are only finitely many vertices => create a cycle
- Algorithm
    - Add 2 integer attributes to every vertex in the graph:
        - label
            - label in the order
        - count
            - initially equals the number of incoming edges (in-degree) of the vertex
            - updated as the algorithm labels vertices
            - always equals the number of incoming edges from vertices **not labelled**
                - require the label of this vertex > all incoming vertices
                - if all vertices that have incoming edges have been labelled, can just label this vertex with a greater value
            - when attribute becomes 0, add vertex to a queue to be labelled
                - any source vertex can be added to the queue immediately
    - Pseudocode:
        -

```
// assume each vertex has 2 integer attributes: label and count
// count is the number of incoming edges from unlabelled vertices
// label will give the topological ordering

for (each vertex v) v.setCount(v.getInDegree()); // initial count values

Set up an empty sourceQueue

for (each vertex v) // add vertices with no incoming edges to the queue
  if (v.getCount() == 0) add v to sourceQueue; // i.e. source vertices

int nextLabel = 1; // initialise labelling (gives topological ordering)
while (sourceQueue is non-empty){
  dequeue v from sourceQueue;
  v.setLabel(nextLabel++); // label vertex (and increment nextLabel)
  for (each w adjacent from v){ // consider each vertex w adjacent from v
    w.setCount(w.getCount() - 1); // update attribute count
    // add vertex to source queue if there are no incoming vertices
    if (w.getCount() == 0) add w to sourceQueue;
  }
}
```

- Correctness
  - A vertex is given a label only when the number of incoming edges from unlabelled vertices is 0
    - all predecessor vertices must already be labelled with smaller numbers
    - dependent on using a queue (FIFO for labelling)
- Analysis (n vertices, m edges)
  - for adjacency lists representation
    - finding in-degree of each vertex is O(n_m)
    - main loop is n times
    - every list is scanned again once
    - Overall: O(n+m)
  - for adjacency matrix representation
    - Overall: O(n^2)

- Deadlock detection
  - Method 1: an adaptation of the topological ordering algorithm
    - if the source queue becomes empty before all vertices are labelled, then there must be a cycle
    - if all vertices can be labelled, then the digraph is acyclic
  - Method 2: an adaptation of DFS
    - when a vertex u is 'visited', check whether there is an edge from u to a vertex v which is on the current path from the current starting vertex
    - the existence of such a vertex indicates a cycle (adaptation of DFS since need to 'remember' current path)

Week 6: Intro to NP Completeness

Polynomial vs exponential time:

- Exponential-time algorithms are in general "bad"
  - increases in processor speeds do not lead to significant changes in the slow behaviour when the input size is large
- Polynomial-time algorithms are in general "good"
- "Efficient algorithms" are in polynomial time
  - Require extra insight
  - Exponential are usually exhaustive
- **Polynomial-time solvable** problem – if it admits a polynomial-time algorithm
  - **Intractable** problem – (proved to be) impossible to find such algorithm
  - NP-complete problem – difficult problem which no one has found a solution for yet (but hasn't proved to be impossible)

NP-complete problems

- No polynomial-time algorithm is known for an NP-complete problem
  - one of them is solvable in polynomial time **=>** they all are
- No proof of intractability is known for an NP-complete problem

    - one of them is intractable **=>** they all are
- Strong belief that NP-complete problems are intractable

Intractable problems

- Causes of intractability:
    - Polynomial time is not sufficient to discover solution
    - Solution itself is so large that exponential time is needed to output it
- Terms:
    - **Undecidable** – no algorithm could solve it
    - Some decidable problems are intractable (can be solved, but not in polynomial time)
- Examples:
    - Roadblock

NP-complete problems

- Problem – characterised by (unspecified) parameters
    - Typically, infinitely many instances for a given problem
- Problem instance – created by giving values to parameters
- Examples:
    - Hamiltonian Cycle (HC)
        - Instance: graph G
        - Question: does G contain a cycle that visits each vertex exactly once?
        - Decision problem – yes/no
            - Every instance is a yes-instance or no-instance
    - Travelling Salesman Decision Problem (TSDP)
        - Instance: a set of n cities and integer distance d(i, j) between each par of cities i, j, and a target integer K
        - Question: Is there a 'travelling salesman tour' of length $\leq$ K
    - Clique Problem (CP)
        - Instance: a graph G and target int K
        - Question: does G contain a clique of size K?
            - A set of K vertices for which there is an edge between all pairs
        - Ex:
            - There is a clique of size 4, but none of size 5
    - Graph Colouring Problem (GCP)
        - Instance: a graph G and target int K
        - Question: Can one of K colours be attached to each vertex of G so that adjacent vertices always have different colours?
    - Satisfiability (SAT)
        - Instance: Boolean expression B in conjunctive normal form (CNF)
        - Question: is B satisfiable?
            - Can values be assigned to the variables that make B true?
- Optimisation and search problems
    - Optimisation problem – find max/min value
        - e.g., the travelling salesman optimisation problem is to find the min length of tour
    - Search problem – find some appropriate optimal structure
        - e.g., the travelling salesman search problem is to find a min length tour

- NP-completeness deals primarily with decision problems
  - Corresponding to each instance of an optimisation/search problem is a family of instances of a decision problem by setting 'target' values
  - Almost invariable, an optimisation/search problem can be solved in polynomial time if and only if the corresponding decision problem can

The class P

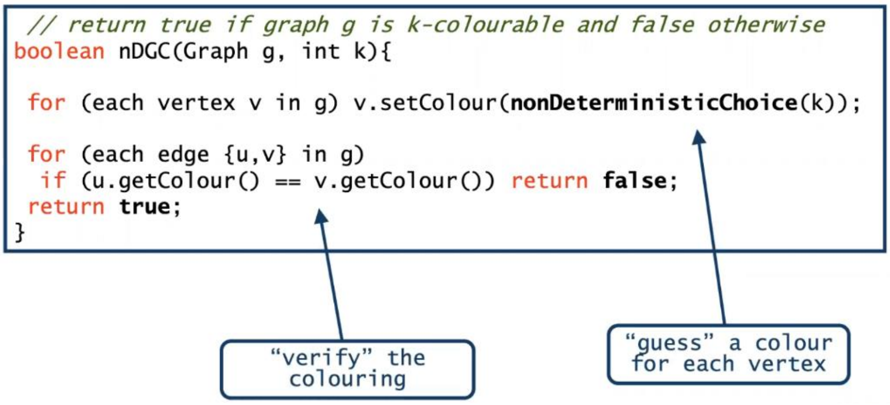
- P – class of all decision problems that can be solved in polynomial time
- Ex:
  - is there a path of length $\leq K$ from vertex u to vertex v in a graph G?
    is there a spanning tree of weight $\leq K$ in a graph G?
    is a graph G bipartite?
    is a graph G connected?
    deadlock detection: does a directed graph D contain a cycle?
    text searching: does a text t contain an occurrence of a string s?
    string distance: is $d(s,t) \leq K$ for strings s and t?
- Often extended to include search and optimisation problems

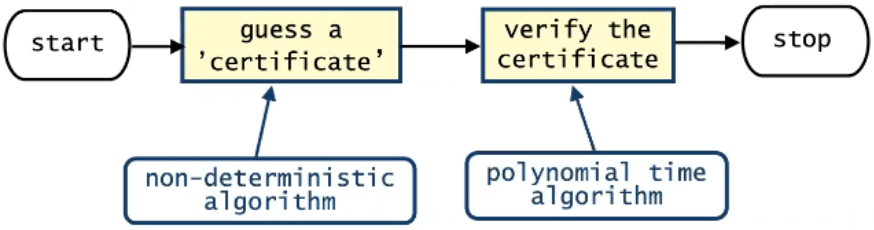
The class NP

- The decision problems solvable in **non-deterministic** polynomial time
  - Non-deterministic algorithm can make non-deterministic choices
    - It's allowed to guess
  - Hence, it's **apparently** more powerful than a normal deterministic algorithm
- P is contained within NP
  - A deterministic algorithm is a special case of a non-deterministic one
  - No problem known to be in NP and known not to be in P
- Relationship between P and NP is most notorious unsolved question (million-dollar prize)

Non-deterministic algorithms (NDAs)

- Such an algorithm has an extra operation: non-deterministic choice
- NDA "**solves**" a decision problem if:
  - for a yes-instance, there is **some** execution that returns yes
  - for a no-instance, there is **no** execution that returns yes
- "solves" a decision problem **in polynomial time** if:
  - for every yes-instance, there is **some** execution that returns 'yes' which uses a number of steps bounded by a polynomial in the input
  - for a no-instance, there is no execution that returns yes
- Not useful in practice
  - But a useful mathematical concept for defining the classes of NP and NP-complete problems
- Ex:
  - Graph Colouring

- 

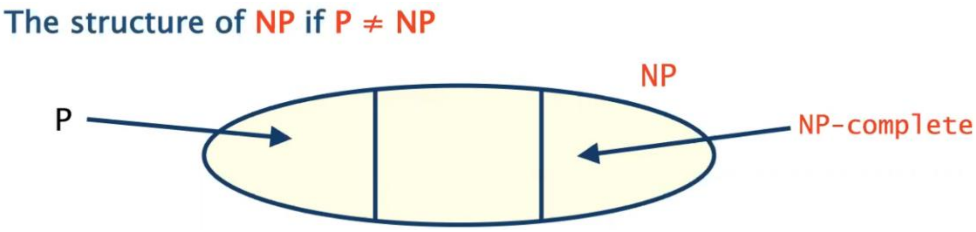
- Structure:
  - Guessing stage (non-deterministic)
  - Checking stage (deterministic and polynomial time)
  - 

Polynomial time reductions

- PTR – mapping f from a decision problem to another decision problem such that
  - for every instance of first DP
    - the instance f(first instance) of second DP can be constructed in polynomial time
    - f(first instance) is a yes-instance of second DP if and only if first instance is a yes-instance of first DP
- There is a polynomial-time reduction from Π_1 to Π_2 – (cut infinity sign)
- Properties
  - Transitivity
    - Pi 1 to Pi 2 and Pi 2 to Pi 3 implies Pi 1 to Pi 3
    - [proof in lecture]
  - Relevance to P: Pi 1 to Pi 2 and Pi 2 in P implies that Pi 1 in P
    - [proof in lecture]

Formal def of NP-completeness

- A decision problem Π is NP-complete if:
  - $\Pi \in NP$
  - For **every** problem $\Pi' \in NP$: Π' is polynomial-time reducable to Π
- Consequences:
  - If Pi is NP-complete and Pi in P, then P=NP
    - Every problem in NP can be solved in polynomial time by reduction to Pi
    - Supposing P $\neq$ NP, if Pi is NP-complete, then Pi is not in P

- 

How to prove a problem is NP-complete

- Suppose know 1 NP-complete problem Π_1
- To prove Pi 2 is NP-complete, enough to show:
    - Pi 2 is in NP
    - There exists a PTR from Pi 1 to Pi 2
- Correctness of approach:
    - For any Pi in NP, since Pi 1 is NP-complete, we have Pi PTR to Pi 1
    - Since Pi PTR to Pi 1 and Pi 1 PTR to Pi 2, it follows that Pi PTR to Pi 2

Cook's Theorem (1971): Satisfiability (SAT) is NP-complete

- the proof consists of a generic PTR to SAT from an abstract definition of a general problem in the class NP
- the generic reduction could be instantiated to give an actual reduction for each individual NP problem
- Given Cook's theorem, to prove a decision problem Pi is NP-complete, it is sufficient to show:
    - Pi is in NP
    - There exists a PTR from SAT to Π

Problem restrictions

- **Restriction** – consists of a subset of the instance of the original problem
    - If a restriction of a given DP Π is NP-complete, then so is Π
    - Given NP-complete problem Π, a restriction of Π **might** be NP-complete
- Examples:
    - Clique restricted to cubic graphs is in P
    - Graph colouring restricted to cubic graphs is NP-complete
    - K-colouring
        - 2-colouring is in P
        - 3-colouring is NP-complete
    - K-SAT
        - 2-SAT is in P
        - 3-SAT is NP-complete (easy to show that SAT $\propto$ 3-SAT
            - [proof in lecture]

Coping with NP-completeness

- Investigate only a **restricted** version of interest (which maybe is in P)
- Seek an exponential-time algorithm improving on exhaustive search
    - e.g., **backtracking**, **branch-and-bound**

    - Should extend the set of solvable instances
- For an optimisation problem,
    - settle for an **approximation algorithm** that runs in polynomial time
    - especially if it gives a provably good result (within some factor of optimal)
    - use a **heuristic**
        - e.g., **genetic algorithms**, **simulated annealing**, **neural networks**
- For a decision problem,
    - settle for a **probabilistic** algorithm (correct answer with high probability)

Week 7: Computability, Finite-State Automata

Introduction

- Computer?
    - input x -> black box -> output f(x)
- Computability concerns which **functions** can be computed
    - "What problems can(not) be solved by a computer?"
    - Need for formal definition
- Unsolvable problems
    - Cannot be solved by a computer even with unbounded time
    - Ex: Tiling Problem
        - Instance: a finite set S of tile descriptions
        - Question: can any finite area, of any size, be completely covered using only tiles of types in S, so that adjacent tiles colour match?
        - No algorithm for this
- Undecidable problems
    - A problem Π that admits no algorithm is called **non-computable** (**unsolvable**)
    - If Π is a decision problem and Π admits no algorithm, it is called **undecidable**
        - e.g., tiling problem
    - Ex: Post's correspondence problem (PCP)
        - Instance 2 finite sequences of words
        - Question: Does concatenating the words in a given sequence give the same results?

The Halting Problem

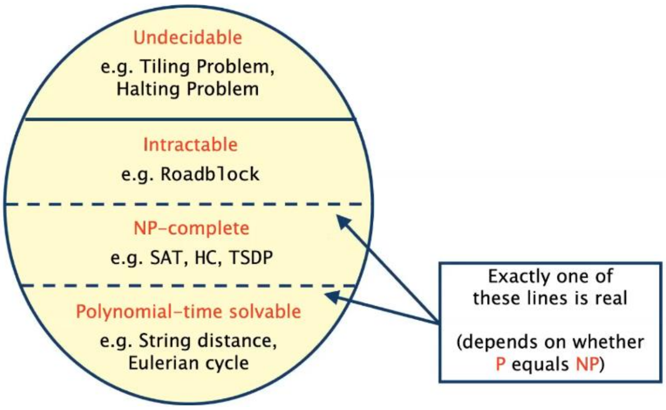
- Impossible project:
    - Write a program Q that takes as input
        - a legal program X
        - an input string S for program X
    - and returns as output
        - yes if program X halts when run with input S
        - no if program X enters an infinite loop when run with input S
- Undecidable
    - Proof
- Formal definition

- Instance: a legal program X and an input string S for X
  - Question: does X halt when run with input S?
  - Theorem: HP is undecidable.
  - Proof by contradiction:
    - [proof in lecture]

Proving undecidability by reduction:

- [proof in lecture]

Hierarchy of decision problems:



Models of computation

- (Attempts to define the "black box")
- **Finite-State Automata (FA)**
  - simple machines with a fixed amount of memory
  - have very limited (but still useful) problem-solving ability
- **Pushdown Automata (PDA)**
  - simple machines with an unlimited memory that behaves like a stack
- **Turing Machines (TM)**
  - simple machines with an unlimited memory that can be used essentially arbitrarily
  - essentially the same power as a typical computer

Deterministic finite-state automata (DFA)

- Simple machines with limited memory which **recognise** input on a read-only tape
- Consists of:
  - a finite input alphabet Σ
  - a finite set of states Q
  - an initial/start state $q_0 \in Q$ and a set of accepting states $F \subseteq Q$
  - control/program or transition relation $T \subseteq (Q \times \Sigma) \times Q$
- Defines a **language**
  - determines whether the string on the input tape belongs to that language
  - in other words, it solves a decision problem (yes if belongs, no if doesn't)
- Recognises/accepts a language
  - the input strings which, when 'run', end in an accepting state

Non-deterministic finite-state automata (NFA)

- Recognition is similar to non-deterministic algorithms "solving" a DP
  - Only require there exists a 'run' that ends in an accepting state
  - i.e., under 1 possible resolution of the non-deterministic choices
- Any NFA can be converted into a DFA (reduction)
  - => Non-determinism does not expand the class of languages that can be recognised by FA
    - being able to guess does not give it any extra power
  - states of the DFA are sets of states of the NFA
  - construction can cause a blow-up in the number of states
    - in worst case, from N states to 2^N states

Regular languages and regular expressions

- **Regular languages** – languages that can be recognised by FA
- A regular language (over an alphabet Σ) can be specified by a **regular expression** over Σ
  - ε (empty string) is an RE
  - σ is an RE
- If R and S are REs, then so are:
  - RS (concatenated)
  - R | S (choice between R or S)
  - R* (0 or more copies of R) (**closure**)
  - (R) (override preference with brackets)

Regular expressions

- Order of precedence (highest first):
  - closure (*), concatenation, choice(|)
  - brackets to override
- Additional operations:
  - Complement ¬x
    - equivalent to the 'or' of all characters except x
  - Any single character ?
    - equivalent to the 'or' of all characters
- Example:
  - No DFA that can recognise strings of the form $a^n b^n$ (number of a's followed by same number of b's)
    - Can be done with some form of memory, e.g., a stack

There are some functions (languages) that are computable, but not by an FA

Week 8: Pushdown Automata

Non-deterministic PDA

- Consists of:
  - a finite input alphabet Σ, a finite set of stack symbols G

- a finite set of states Q including start state and set of accepting states
- control or transition relation $T \subseteq (Q \times \Sigma \cup \{\varepsilon\} \times G \cup \{\varepsilon\}) \times (Q \times G \cup \{\varepsilon\})$
  - ε – empty set

- Accepts an input if and only if, after the input has been read, the stack is empty and control is in an accepting state
- There is no explicit test that the stack is empty
  - This can be achieved by adding a special symbol ($) to the stack at the start of the computation
  - i.e., we add the symbol to the stack when we know the stack is empty and never add $ at any other point
  - Check for emptiness by checking if $ is on top
- Deterministic PDAs (**DPDAs**) are **less powerful**
  - There are languages that can be recognised by NDPDA but not by DPDA (e.g., language of palindromes)
- **Palindromes**
  - How to recognise:
    - Push first half onto stack
    - As we read each new char, check it is the same as top element and pop this
    - Enter an accepting state if all checks succeed
  - Why **non-determinism**?
    - Need to "**guess**" where the **middle** is
      - and if there's even/odd chars
    - Cannot work this out first and then check the string, would need:
      - to read string twice
      - an unbounded number of states if string is infinite
  - Automaton recognises $\{ a\hat{}\{n\}\ b\hat{}\{n\} \mid n \geq 0 \}$
    - => PDAs are more powerful than FAs
- Languages that can be recognised by a PDA are **context-free languages**
  - Are all languages context-free?
    - No, consider $a^n b^n c^n$

Turing machines

- T to recognise a particular language consists of:
  - a finite alphabet Σ, including a blank symbol (#)
  - an unbounded tape of squares
    - each can hold a single symbol
    - tape unbounded in both directions
  - a tape head that scans a single square
    - can read from it and write to the square
    - then moves one square left/right along the tape
  - a set S of states
    - includes a single start date s_0 and 2 halt (terminal) states s_Y and s_N
- Transition function:
  - $f: \left((S/\{s_Y, s_N\}) \times \Sigma\right) \rightarrow (S \times \Sigma\{Left, Right\})$
  - For each non-terminal state and symbol, the function f specifies:
    - a new state (perhaps unchanged)
    - a new symbol (perhaps unchanged)

        - a direction to move along the tape
    - $f(s, \sigma)(s', \sigma', d)$ means reading symbol σ from the tape in state s
        - mot to state $s' \in S$
        - overwrite the symbol σ on the tape with the symbol σ′ ∈ Σ.
        - move the tape head one square in direction d (left/right)
- Computation
    - The (finite) input string is placed on the tape
        - assume initially all other squares of the tape contain blanks
    - The tape head is placed on the first symbol of the input
    - T starts in state s0 (scanning the first symbol)
        - If T halts in sY, ‘yes’
        - Otherwise, ‘no’
- Palindrome problem
    - Instance: a finite string Y
    - Question: is Y a palindrome, i.e., is Y equal to the reverse of itself?
    - [lecture]
- A TM can be described by its state transition diagram – directed graph where:
    - each state is a vertex
    - …
- The TM that accepts language L actually computes the function f where f(x)=1 if x is in L, 0 otherwise
    - Definition of a TM can be amended as follows:
        - to have a set H of halt states
        - the function it computes is defined by f(x)=y where:
            - x – initial string on tape
            - y – string on tape when machine halts
- Language L is **Turing-recognisable** if some TM **recognises** it, i.e., given an input string x:
    - if x is in L, then the TM halts in state sY
    - if x is not in L, then the TM halts in state sN **or fails to halt**
- Language L is **Turing-decidable** if some TM **decides** it, ie., given input string x:
    - if x is in L, then the TM halts in state sY
    - if x is not in L, then the TM halts in state sN
- Every decidable language is recognisable, but not every recognisable language is decidable
    - e.g., Halting Problem language
- Enhanced Turing machines
    - 2+ tapes
    - 2D tape
    - non-deterministic
    - (none of these change computing power)
        - Proof: any enhanced TM can be **simulated** by basic TM
- P and NP
    - P is class of DPs solvable by a TM in polynomial time
    - NP is solvable by non-deterministic TM in polynomial time
        - Non-determinism changes only speed of computation

Counter programs

- Completely different model of computation:

    - All general purpose programming languages have essentially the same computational power
- Have:
    - variables of type int
    - labelled statements of form:
        - L : unlabelled_statement
    - Unlabelled statements of the form:
        - x = 0
        - x = y + 1
        - x = y – 1
        - if x==0 goto L
        - halt;

Church-Turing Thesis

- The TM is an appropriate model for the 'black box'
    - A whole range of different computational models are equivalent in their abilities

Week 9: Revision

Exam format:

- 4 questions
- 60 marks (15 for each question)
- Duration: 1h 30 mins
    - (extra 30 mins for "uploading")
    - allocate ~20 mins per question
- Examinable material – everything in lectures except:
    - correctness proof for Prim-Jarnik
    - reduction from SAT to Clique
    - proof halting problem is undecidable
    - tutorial questions marked as hard
- Objective: test understanding, not memory
    - **must use own words**
- Advice:
    -
    - if asked to 'describe algorithm X', convince examiner you understand by using a mix of English and pseudocode, possibly with illustrative example
    - Allocation of marks will suggest level of detail expected
    - No own words, no marks
    - Word doc, but draw on paper and insert image/scan into the doc

Problem and problem instances:

- Problem – usually characterised by (unspecified) parameters
- Problem instance – created by giving these parameters values

Class NP

- Decision problems solvable in non-deterministic polynomial time
  - non-deterministic algorithm can make non-deterministic choices
    - apparently more powerful than a normal deterministic algorithm
  - NDA has many possible executions depending on choices made
- NDA:
  - NDA 'solves' a decision problem Pi if:
    - for a 'yes' instance, there is **some** execution that returns 'yes'
    - for a 'no' instance, there is **no** execution that returns 'yes'
  - NDA 'solves' in polynomial time if:
    - … (see earlier lecture)

Polynomial time reductions

- A PTR is a mapping f from a decision problem Pi1 to a DP Pi2, such that, for every instance i1 of Pi1, we have:
  - the instance f(i1) of Pi2 can be constructed in polynomial time
  - f(i1) is a 'yes' instance of Pi2 if and only if i1 is a 'yes' instance of Pi1
- PTR from Pi1 to Pi2:

$$\Pi_1 \propto \Pi_2$$

NP-complete

- DP Pi is NP-complete if:
  - Pi is in NP
  - for every problem Pi' in NP, Pi' is polynomial time reducible to Pi
- Consequences of definition
  - If Pi is NP-complete and Pi is in P, then P=NP
    - every problem in NP can be solved in polynomial time by reduction to Pi
  - If P=/=NP and Pi is NP-complete, then Pi is not in P
- Proving:
  - To prove Pi2 is NP-complete, enough to show:
    - Pi2 is in NP
    - there exists a PTR from Pi1 to Pi2
  - How to prove PTR:
    - Figure out which direction the reduction needs to go
      - Suppose we need from Pi1 to Pi2
    - Next, build it:
      - given an instance i1 of Pi1, construct and instance of f(i1)=i2 of Pi2
      - if there is a hint, use it, and otherwise think about how they're related
      - write down/draw instances side by side
      - relate parameters if there are any
    - Finally, show it's correct:
      - Show you can perform construction in polynomial time
      - … [watch recording] ???

Building automata and Turing machines

- For DFAs, go through each possible option (action) in each state
    - moving to a new state if you have progressed towards the language
    - and moving to an accepting state or sink state if appropriate
- For PDAs, first think about the roles of the stack as a form of memory and then follow a similar approach to DFAs updating the stack
- Defining a TM for even simple problems is hard
    - (won't be asked to make a diagram)
    - Write pseudocode