Overview of Sections 2.2, 3.1-3.3, 4.1, and 6.1-2

March 31, 2022

Abstract

Your test will resemble the problems from your homework assignments, and problems from previous tests I've given. You will probably have 7 equally weighted questions or so (one every ten minutes!). Attempt every problem (don't get bogged down trying to make one **perfect**, while ignoring others that you could get some points for). Don't let the perfect be the enemy of the good....

1 Section 2.2

Induction is a proof technique which is useful for demonstrating a property ($P(n)$) for an infinite ladder of propositions (think of our property as being indexed by n, as in $P(n)$. Induction begins with a base case (or an anchor) and then proceeds via an inductive case (often $P(k) \rightarrow P(k+1)$).

I like to say that this is just dominoes falling (or "infinite *modus ponens*"): we know $P(0)$ (the first domino falls), and that $P(k) \rightarrow P(k+1)$ (if the kth domino falls, so does the (k+1)th domino). Therefore we know $P(1)$; and so on. All the dominoes are falling down.

There are two different (but equivalent) principles of induction, the first and second. The second appears to assume more than the first: the inductive hypothesis in the second principle is that the property is true for all cases up to and including the k^{th} case.

2 Sections 3.1-3.3

Recursion in section 3.1 looks very much like induction: the idea is that we have a base case (or cases), and from there we generate additional cases. Unlike induction, the set of things we generate may not be easily indexed to the integers. For example,

- the palindromes on binary strings, or
- the construction of wffs using the logical connectives.

In this section we see how to solve one particular recurrence relation, the linear, first-order, constant-coefficient recurrence relation.

Once we deduce this formula (and prove it by induction), we needn't ever solve another linear, first-order, constant-coefficient recurrence relation from scratch: we can just invoke the formula. This is our quest, the holy grail!

The general method is to "guess, check, and verify": compute some terms, look for a pattern, characterize the pattern, then prove it.

A different variety of recurrence relation occurs in the analysis of algorithms, when we consider "divide and conquer" algorithms (such as *BinarySearch*).

By changing variables, we can get a closed form solution for the number of operations for these "divide and conquer" algorithms.

In the analysis of algorithms we are interested in efficiency, and will count operations in order to compare competing algorithms. We can sometimes count operations directly, but may resort to recursion to count. We're interested in the worst-case scenario.

3 Section 4.1

- The notation of sets (definition, order, cardinality, empty set, power set, Cartesian products, countable, uncountable, ...)
- Using predicate logic to determine when two sets are equal

- Subsets, proper and improper; the set of all subsets of a set is the power set (always bigger than the set itself).
- The unusually useful empty set, denoted $\oslash$ or $\{\}$, a subset of every set.
- Binary and unary operations (and conditions for their proper definition). So, conditions for a binary operation: for every ordered pair of elements of the set (domain), the operation gives a result – an "image" – that 1) exists, 2) is unique, and 3) is an element of the set (closure))
- Intersection, union, and set-difference are binary operations on the power set of a set (complements are unary). Venn diagrams display them.
- Pascal's Triangle is useful for envisioning the various subsets of a given size (a row of the triangle shows you the different subsets of each size).
- one-to-one correspondence, and proving that cardinalities are the same (i.e., that sets have the same size)
- Infinite sets – that there are infinitely many different sizes of infinity, proven by means a diagonalization-like attempt to map the power set of a set A to itself (to A). (Power sets are incredibly powerful!)

4 Section 6.1

- Graph definitions and terminology
- Special graphs (K_n, $K_{m,n}$), and special graph properties (directed graphs, trees, chains, simple, complete, connected, etc.)
- Isomorphic graphs:
  - **Definition**: **Bijection**. A bijection is a one-to-one and onto mapping from set A to set B – every element of **each** set has a unique partner in the other set.

  - **Definition**: Two graphs (N_1, A_1, g_1) and (N_2, A_2, g_2) are **isomorphic** if there are bijections $f_1 : N_1 \to N_2$ and $f_2 : A_1 \to A_2$ such that for each arc $a \in A_1$, $g_1(a) = \{x, y\} \iff g_2[f_2(a)] = \{f_1(x), f_1(y)\}$ (replace braces by parentheses for a directed graph).
  - **Theorem**: Two simple graphs (N_1, A_1, g_1) and (N_2, A_2, g_2) are isomorphic if there is a bijection $f : N_1 \to N_2$ such that for any nodes n_i and n_j of N_1, n_i and n_j are adjacent $\iff$ $f(n_i)$ and $f(n_j)$ are adjacent.
  - Tests for when two graphs are **not** isomorphic (or "non-isomorphic") – any intrinsic property that can fail provides a test (e.g. different number of nodes, edges, three-cycles; one connected, one not; etc. in the two graphs).

- Planar graphs (one which can be drawn in two-dimensions so that its arcs intersect only in nodes)

- Euler's Formula for connected planar graphs states that

$$r - a + n = 2$$

  ("ran2") where n is the number of nodes, a is the number of arcs, and r is the number of regions (including the infinite region surrounding the graph).

- A non-planar, connected graph contains a copy of a graph essentially isomorphic to K_5 or $K_{3,3}$. In particular this proves that every connected graph with fewer than 5 nodes is planar!

- Computer representations of graphs:
  - the adjacency matrix, and
  - the adjacency list.

  (advantages of one over the other).

5 Section 6.2

- **tree**: an acyclic, connected graph with one node designated as the **root** node – or defined recursively as
  - **Base case**: One node is a tree, itself the root.
  - **Inductive step**: The graph formed by attaching a new node r by a single arc to each root of n trees $\{T_i\}_{i \in \{1,\dots,n\}}$ is a tree.
- tree terminology
- examples of trees (especially binary)
- tree representations (every tree is a graph, so we can use graph representations; but there are tree-specific representations, too).
- tree traversal algorithms (make sure to sing these as you do them!:):

preorder	*root*	*left*	*right*
inorder	*left*	*root*	*right*
postorder	*left*	*right*	*root*