Table of contents

1. [Exam 1 content](#ca1nfmeptw0h)
2. [Exam 2 content](#soakuujpf4p1)
3. Post-exam 2 content
4. [terms](#94svxqwb49qx)

Discrete Math study guide - Exam 1

**Chapter 1 - Propositions, operators, and proofs**

Types of propositions:

| p | q | Conjunction - p ∧ q | Disjunction - p ∨ q | Exclusive OR - p ⊕ q | Conditional -  p → q | Biconditional -  p ↔ q |
| --- | --- | --- | --- | --- | --- | --- |
| T | T | T | T | F | T | T |
| T | F | F | T | T | F | F |
| F | T | F | T | T | T | F |
| F | F | F | F | F | T | T |

Unique types of conditional based around p → q:

* contrapositive: ¬q → ¬p {has the same truth value as p→q}
* Converse: q → p {doesn’t have same truth value, equivalent to Inverse}
* Inverse : ¬p → ¬q {doesn’t have same truth value}

Full definition of biconditional : p ↔ q ≡ (p → q) ∧ (q → p)

Expression: “p iif q”, “p exactly when q”, “p is necessary and sufficient for q”

Logical equivalent = p ☰ q if p ↔ q is a tautology

Important logical equivalences:

| Identity law | p ∧ T ≡ p **||** p ∨ F ≡ p | De Morgan’s law | ¬(p ∧ q) ≡ ¬p ∨ ¬q **||**  ¬(p ∨ q) ≡ ¬p ∧ ¬q |
| --- | --- | --- | --- |
| Domination law | p ∨ T ≡ T **||** p ∧ F ≡ F | Conditional disjunction | p → q ≡ ¬p ∨ q |
| Double negation | ¬(¬p) ≡ p | Distributive laws | p ∨ (q ∧ r) ≡ (p ∨ q) ∧ (p ∨ r)  p ∧ (q ∨ r) ≡ (p ∧ q) ∨ (p ∧ r) |
| Commutative law |  | Associative law |  |
| Absorption laws | p∨(p∧q)≡ p  p∧(p∨q)≡ p |  |  |

Satisfiability -

* Satisfiable = there is some assignment of truth values to make a compound proposition **T** (a tautology / normal proposition)
* Unsatisfiable = there exist no assignment of truth values to make a compound proposition **T** (a contradiction)

Predicates / Quantifiers:

Propositional function = P(X) : Predicate(x)

* Predicate is some manipulation of variable x

Types of quantifiers:

| **Universal quantifiers** | ∀xP(X) | P(X) is true for all x || there exists one x where P(X) = **F** | Associated with ∧ |
| --- | --- | --- | --- |
| **Existential quantifiers** | ∃xP(X) | There exists one x where P(X) = **T** || P(X) is false for all x | Associated with ∨ |

* Note - quantifiers should only bind up to 1 variable ⇒ ∀xP(X + Y) [X is bound, but Y is free ]
* Only when all variables are bound can you form a proposition from a propositional function (narrow a function into either true or false)

Important predicate equivalents

* ∀x(P(X) ∧ Q(X)) = ∀x(P(X)) ∧ ∀x(Q(X)) [this can be generalized by using any combination of quantifiers and operators]
* ¬∀xP(x) ≡ ∃x ¬P(x) || ¬∃xP(x) ≡ ∀x ¬P(x).
  + This can be proven by writing out the proposition related to either quantifier and applying de Morgans law to them

Nested Quantifiers:

∀x∀yP(x, y) = for every (x,y) pair, P(x,y) = **T**

∀x∃yP(x, y) = for every x, there exists 1 assignment of y where P(x,y) = **T**

∃x∀yP(x, y) = there exists 1 assignment of x where every possible y P(X,y) = **T**

∃x∃yP(x, y) = there exists an (x,y) pair where P(x,y) = **T**

Negating nested quantifiers : ¬∀x∃yP(x,y) = ∃x¬∃yP(x,y) = ∃x∀y¬P(x,y)

Argument form:

P (p (p→q)) → q = **T**

p → q ⇒

\_\_\_\_\_

∴ q

Types of proofs for p → q:

* **Direct proof** = assuming p is true ⇒ prove that q is true
* **Proof by contraposition** = assuming ¬q is true ⇒ prove that ¬p is true (¬q->¬p)
* **Proof by contradiction** = assume ¬p and q are true ⇒ try to show that this assumption leads to some contradiction (like both p and ¬p are true)(¬p->q)
* **Vacuous proof** = show that p is false ⇒ p→q must be true
* **Trivial proof** = show that q is true ⇒ p→q must be true
* **Proof of cases** = prove a few representative cases ⇒ prove overall theorem
* **Exhaustive proof** = prove each possible case ⇒ prove overall theorem

Existence proof: trying to prove ∃xP(x)

* Constructive = find some *a* where P(a) = **T**
* Non-constructive = prove that ∃xP(x) = **T** without showing that P(a) is true for some a (example pg 102)

Uniqueness proof: trying to prove that there is exactly one element with some said property → if x and y both have the desired property, then x = y

**Chapter 2 - Sets, functions, sequences, sums and Matrices**

Set = an unordered collection of *distinct* objects (elements/members of set)

Element a belongs to set A = a ∈ A

*Ways to write out a Set*:

* Roster: A = {a,b,c,d}
* Set builder: A = {x ∈ **+Z** | x <10 } = {1,2,3,4,5,6,7,8,9}
* not equalNull set : E = {∅}
* 2 sets are equivalent if: ∀x(x ∈ A ↔ x ∈ B) [ A and B share the same members]
* E.g. {1,3,3,3,3,5,5,5,5} is same as {1,3,5}

Subset = set A is a subset of set B only if every member of set A is also a member of set B

* A ⊆ B (which is not equal to B ⊆ A)
* To show subset, show that every member in A is a member of B
* You can use subsets to prove equivalence: prove A ⊆ B and B ⊆ A → A = B
* Every set S has these 2 properties:
  + ∅ ⊆ S
  + S ⊆ S

Cardinality: | S | = the number of distinct members

* 2 sets have the *same cardinality* iff set A → set B = bijective + | A | = | B |

Power set: *P*(S) = the set of all subset of set S

* | *P*(S) | = 2 ^ (| S |)

Ordered n-tuples: (a1,a2,a3,...,an) an ordered collection of elements where a1 is 1st and an is last

* Only equivalent if 2 tuples have both the same members + same order

Cartesian product: defined in terms of two sets A, B ⇒ A x B = {(a, b) ∣ a ∈ A ∧ b ∈ B} [set of all ordered pairs (a,b) where a ∈ A and b ∈ B ]

* A x B ≠ B x A unless A = B

Set operators: working with set A and B, set operators generally create another set

* Union: A ∪ B = set contains all elements from A or B or both
* intersection : A ∩ B = set contains only elements from A and B
  + Disjoint sets = where S1 ∩ S2 = ∅
* Difference: A - B = set of elements in A but not in B
  + A - B ≠ B - A
* Complement = ㄱA = elements not in A = U - A

Important set laws:

| Identity law | A ∩ U = A || A ∪ ∅ = A | De Morgan’s law | ㄱ(A ∩ B) = ㄱA ∪ ㄱB |
| --- | --- | --- | --- |
| Complementation law | ㄱ(ㄱ(A)) = A | Complement law | A ∪ ㄱA = U || A ∩ㄱ A = ∅ |
| Distributive law | A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C) |  |  |
| Commutative law |  | Associative law |  |

Some ways to prove set identity:

* Subset method = show that both sides of identity are subsets of each other
* Applying existing identities = transform 1 side of the identity into the other

Multisets: an unordered collection of elements where elements can occur multiple times

* As opposed to sets where all elements should be distinct
* Represented as: {m1 \* a1 , m2 \* a2 , … mn \* an }
* *Multiplicities (mi)* - the # of times an element is repeated
  + All elements not in the multiset have a multiplicity of 0
* *Union of multisets* - out of multiSets M and P ⇒ take the largest multiplicity term for the same element
* *Intersection of multisets* - take the minimum multiplicity term for the same elements
* *Difference of multisets (P- M)*  - (the multiplicity of element ai in P ) - (the multiplicity of ai in M) for every element in both P and M
* *Sum of multisets (P+M)* - the opposite of difference

Function: for 2 non-empty sets A,B → function from A → B is the assignment of exactly 1 element of B to each element of A (no element in A should not have an assignment)

* Domain = set A | codomain = set B
* 2 functions are the same when functions have: same domain, codomain, and the same mapping of domain to codomain for every element in the domain

Types of functions

* **One-to-one (injective)** = no element in the codomain is assigned to more than once: if f(a) = f(b) then a = b
  + To prove not injective = find elements x,y in domain where x ≠ y but f(x) = f(y)
* **Onto (surjective)** = for every element b in the codomain, there is an element a in the domain so that f(a) = b
  + To prove not surjective = find some element in codomain that has no element in the domain to match
* **one-to-one correspondence(bijection)** = both one-to-one and onto

Increasing and decreasing functions: only applicable if domain + codomain ⊆ **R**

* Increasing: f(x) ≤ f(y) when x < y | Strictly increasing: f(x) < f(y) when x < y
* Decreasing: f(x) ≥ f(y) when x < y | Strictly decreasing: f(x) > f(y) when x < y

Inverse function: **only for bijective functions**, where the mapping went f(a) = b, the inverse would be f-1(b) = a

Composite function: f ο g(x) = f(g(x))

Identity of a function = f o f-1(b) = b where b is an element in the codomain | f-1 o f(a) = a where a is an element in the domain

Partial function: defined from A → B, only a subset of A has a an assignment (domain of definition), elements outside of the domain are undefined

Sequence: function from some subset of **Z** → set S, an - denotes the n-th term in the sequence

* 1 way to define a sequence: an = 1/n ⇒ 1/1,½, ⅓, ¼, … , 1/n

Types of sequences

* **Geometric progression**: ar0, ar1, ar2, ar3 … {a is the initial term, r is constant ratio}
* **arithmetic progression**: a + 0b, a+1b, a+2b, a+3b, …
* **Recurrence relation**: an is defined in terms of previous terms + requires some base case (initial condition)

Useful summation formulas:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Matrices: represented like A = [ai,j ] denotes a m X n matrix named A

* Matrix addition: both matrices need to be the same size ⇒ A + B = for every element (i,j) = ai,j + bi,j
* Matrix multiplication: matrix A = m X k | B = k X n ⇒ **A** X **B** = [ci,j]
  + Ci,j = (a1,j \* bi,1) + (a2,j \* bi,2) + (a3,j \* bi,3) + … + (ak,j \* bi,k)

Identity Matrix: an n X n matrix where for Iij : when i = j Iij= 1 , when i ≠ j Iij = 0

Inverse matrix: the inverse of matrix **A** is a matrix **B** of the same size (n x n) where **AB** = **BA** = **In** (the identity matrix for the that size)

Boolean product of a matrix: like matrix multiplication, but checking the 0,1 values:

* Ci,j = (a1,j ∧ bi,1) ∨ (a2,j ∧ bi,2) ∨ (a3,j ∧ bi,3) ∨ … ∨ (ak,j ∧ bi,k)

**Chapter 3 - Algorithms**

Searching Algorithm

Linear search(sequential search)

Pseudocode:

i:= 1

While (i<= n and x!= ai)

i:= i+1

If i<=n then location := i

Else location := 0

Return location(0 if x is not found)

Binary search

Pseudocode: (find x)

i := 1 (left endpoint of search interval)

j := n (right endpoint of search interval)

While i< j

M := ⌊(i + j)/2⌋

If x > am then i := m+1

Else j:= m

If x=ai then location := i

Else location := 0

Return location(0 if x is not found)

Sorting

Bubble Sort(a1,…, an : real numbers with n ≥ 2)

Pseudocode:

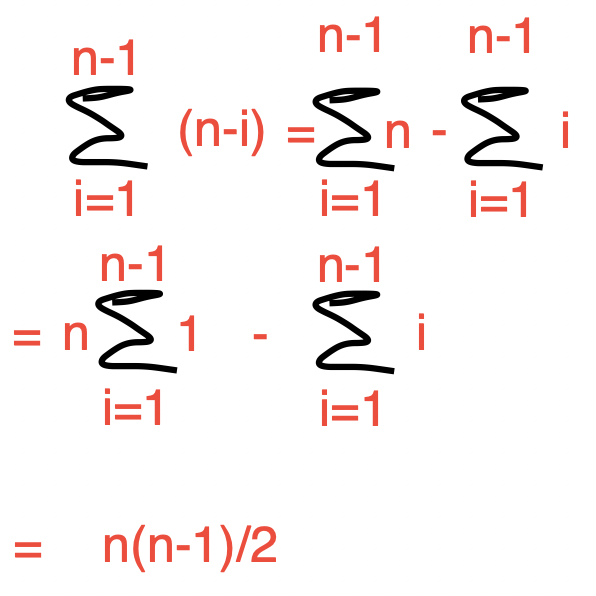
For i:= 1 to n-1

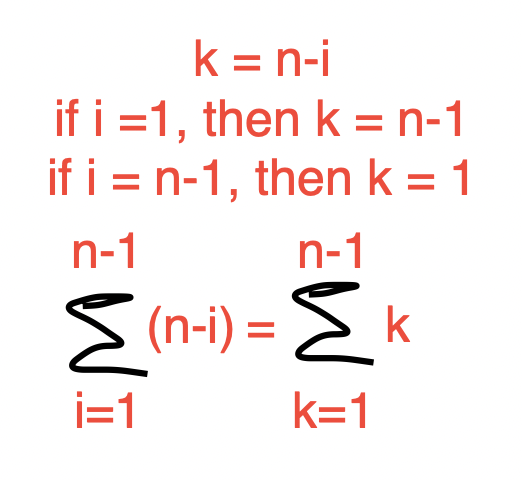
For j := 1 to n-i

If aj > aj+1 then interchange aj and aj+1

{a1, …, an is in increasing order}

Refer to Homework4





**Big-O Notation**

Definition: f(x) is O(g(x)) if there are constants C and k such that |f(x)|<= C|g(x)|, whenever x > k

(C and k are called witnesses)

E.g. f(x) = x^2 +2x+1

Assume k=0, c=2

For every x >= 0, x^2 + 2x + 1 <= 2x^2, so x^2 >= 2x+1

When x = 2, it is false; when x=3, it is true.

So k=3, namely, for every x >= 3

To show A is O(b) → find a pair of C and k

To show A is not O(b) → no pair of witnesses C and k exist such that A <= Cb whenever x>k. Use a proof by contradiction to show it. (Suppose C and k exist. No matter what C and k are, the inequality cannot hold for all x > k )

**Big-Omega and Big-Theta Notation**

**Lower bound: Big-Omega**

f(x) is Ω(g(x)) if there are constants C and k with C positive such that |f(x)| >= C|g(x)| whenever x > k.

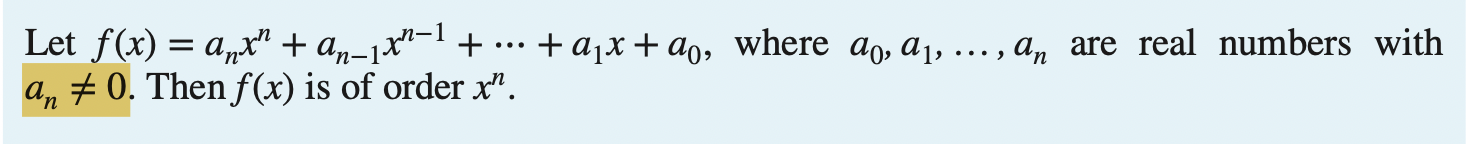
f(x) is Ω(g(x)) if and only if (iif) g(x) is O(f(x))

**Big-Theta**

f(x) is θ(g(x)) if f(x) is O(g(x)) and f(x) is Ω(g(x)). f(x) is of order g(x), and that f(x) and g(x) are of the same order.

C1|g(x)| <= |f(x)| <= C2|g(x)|, whenever x > k.

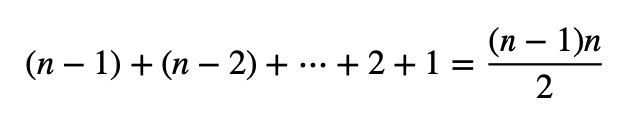
To show the polynomial f(x) is Big-theta of g(x). → 1. f(x) is O(g(x)) 2. g(x) is O(f(x))



**Time Complexity**

Bubble Sort:

The worst-case complexity: θ(n^2)



**Complexity of Matrix Multiplication**

Use the least number of multiplications of integers:

A1: 30x20, A2:20x40, A3:40x10. The order: A1(A2A3)

**Brute-force Algorithm**

Exponential time

E.g. find the maximum number in a sequence, bubble, insertion and selection sorts

| Commonly Used Terminology for the Complexity of Algorithms | |
| --- | --- |
| Complexity | terminology |
| θ(1) | Constant complexity |
| θ(logn) | Logarithmic complexity |
| θ(n) | Linear complexity |
| θ(n^b) | Polynomial complexity |
| θ(b^n), where b > 1 | Exponential complexity |
| θ(n!) | Factorial complexity |

**Tractability**

A problem that is solvable using an algorithm with polynomial (or better) worst-case complexity is called **tractable**, because the expectation is that the algorithm will produce the solution to the problem for reasonably sized input in a relatively short time.

The situation is much worse for problems that cannot be solved using an algorithm with worst-case polynomial time complexity. Such problems are called **intractable**.

P Versus NP

Problems for which a solution can be checked in polynomial time are said to belong to the class NP (tractable problems are said to belong to class P)

|  | P | NP  (nondeterministic polynomial) |
| --- | --- | --- |
| Easy to solve | yes | no |
| Easy to verify | yes | yes |

**Chapter 9 - Relations (599 - 601 | 606 | 609)**

A binary relation from A to B is a subset of A X B

A relation on a set A is a subset of A x A

R1 - R2: (x, y) is in R1 but not in R2

R1⨁R2 = R1 U R2 - R1R2

S o R same as f o g = f(g(x))

Page 609

Let *R* be a relation from a set *A* to a set *B*. The **inverse relation** from *B* to *A*, denoted by *R*−1, is the set of ordered pairs {(*b, a*) ∣ (*a, b*) ∈ *R*}. The **complementary relation** *R* is the set of ordered pairs {(*a, b*) ∣ (*a, b*) ∉ *R*}.

**Chapter 12 - Boolean Algebra (853)**

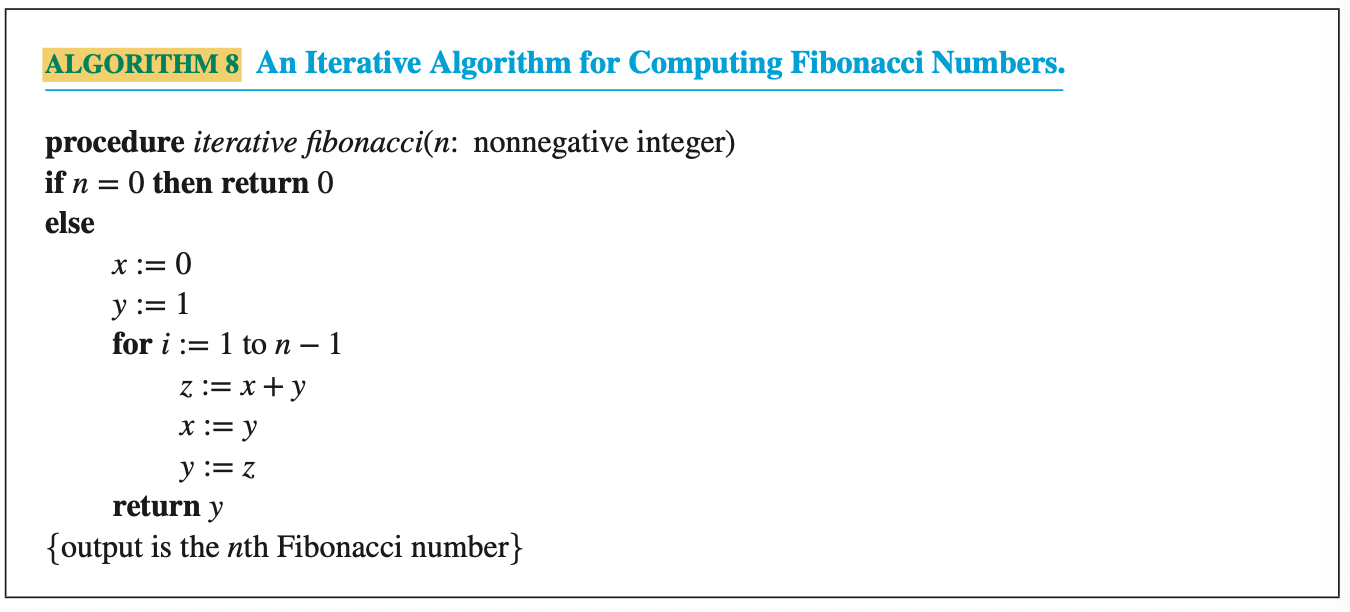
Boolean Algebra Laws:

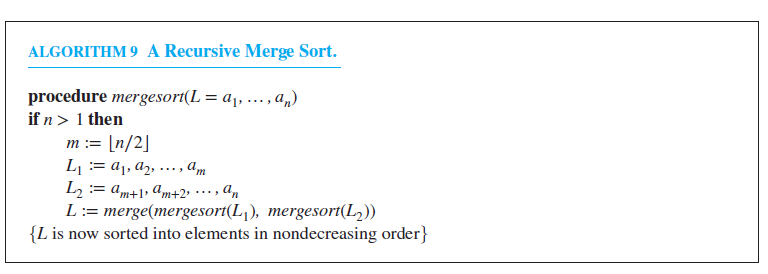
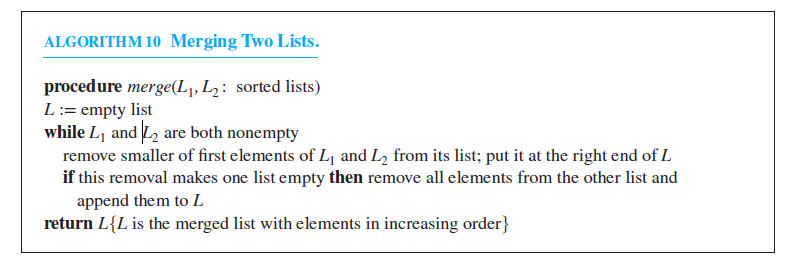
| Identity Laws |  |
| --- | --- |
| Complement Laws | = 1 |
| Associative Laws |  |
| Commutative Laws |  |
| Distributive Laws |  |

Discrete Math - Exam 2 (chapter 5,6,9,10)

**Chapter 5** - induction + recursion

* *Mathematical induction proves if P(n) is true for all non-negative integers* ***n***
* Basic setup of a mathematical induction:
  + Basis step: very that P(1) or P(0) is true [based on whether P(L) where L is the lowest possible value for the proposition]
  + Inductive Step: assume P(k) is true *(inductive hypothesis)* then prove P(k) → P(k+1)
* Well ordering principle = in every nonempty subset of the non-negative set of integers, there must be a least element
  + This is why induction allows you to prove something for all non-negative integers
* Induction vs strong induction:
  + Basis step is the same in both
  + Inductive step: assume that P(1) Λ P(2) Λ P(3) … Λ P(k) is true: prove P(1) Λ P(2) Λ P(3) … Λ P(k) → P(k+1)
* Recursively defined function: similar setup to inductive proofs:
  + Basis case: specify the value of the function at some minimum value
  + Recursive step: give the rule for finding a value from smaller values
* Recursive functions are well-defined when every positive integer has a unambiguous image in the recursive function
* Sets can also be recursively defined:
  + Basis step: which elements are part of the set at start
  + Recursive step: how to generate new elements of the set
* Special set of note: set Σ∗ of strings over the alphabet Σ
  + Base case: 𝜆 ∈ Σ\* (Where 𝜆 = empty string = “”)
  + If w ∈ Σ\* and x ∈ Σ [if w in the set of strings AND x is a char] then wx ∈ Σ\*
* Concatenations (with operator: \*) are also defined recursively
  + Base case: If w ∈ Σ∗, then w ⋅ 𝜆 = w, where 𝜆 is the empty string
  + Recursive step: If w1 ∈ Σ\* and w2 ∈ Σ\* and x ∈ Σ, then w1 ⋅ (w2x) = (w1 ⋅ w2)x.
  + [if w1 and w2 belong to the set of strings and x is some char {basically make x is the last character in string 2} → it is possible to concatenate string 1 and string 2]
* Structural induction: a form of induction used to prove theorems about recursively defined sets. P376
  + Basis step: prove that theorem == true for basis cases
  + Recursive step: show that theorem == true for elements used to construct a new element:
  + This differs based on what you are trying to prove:
    - Proving compound proposition: prove that (¬p), (p ∨ q), (p ∧ q), (p → q), and (p ↔ q) are all TRUE
    - Proving normal proposition: just prove P(k)
* Review the logic for Professor’s logic for pg 378 found in lecture from 10/8
* Recursive algorithm = solves problems by reducing problem into same problem but an input closer to desired result
* Recursive algorithms generally take more computing power than iterative approach but is easier to write
  + Ex: fibonacci → recursive = O(n2) , iterative = O(n)\
* Hw 6 operators - a | b = a divides b = b/a is an integer
* Hw 6 operators - **A** = some matrix → **A**n = **An-1** X **A →**  **A**n =A x **An-1**



* Merge sort: given a list of n numbers
  + Split the list of numbers into sublists half the size of the original list until sublist have lengths of 0 or 1. O(logn)
  + Then merge the sublist together: 1 pointer in one list, 1 pointer in the other list → create a new list by inserting the value of the smaller pointer then iterating that pointer. O(n)
  + If you empty out 1 sublist, just add all the elements of the other list
  + Do this with all the sublist until you end up with the original list
  + Is O(n log n) complexity
* Merge sort pseudo code:
  + 
* Merge pseudocode :
  + 
* Resulting lemma = 2 sorted lists m and n, will take no more than |m| + |n| - 1 to be merged together
* The number of comparisons needed to merge sort a list with n elements is O(nlogn).

**Math for CS - chapter 2** - Well-ordering proof

* Logic for proving a number is rational = value can be represented by 2 integers in the form of m/n and can be written in lowest terms where m and n have no common prime factors
  + Assume that m/n cannot be written in lowest terms → this allows you to construct set C = all the non-negative integer numerators that would satisfy m/n
  + By the well-ordering property, there is some element in C that is the least, call this m0.
  + Since both m and n are non-negative integers, n0 > 0
  + Because m0 / n0 cannot be written lowest terms with no common prime factors, there is some common factor, p > 1, that exists → (m0 / p) / (n0 / p)
  + Since (m0 / p) < m0, m0 is not the least element which is a contradiction to the well-ordering property → proposition is true
* Proving propositions via the well-ordering property usually takes the form of this:
  + Trying to prove: P(t) holds for every non-negative integer
  + Define a set C which are all the examples of P being false: C = {n∈ **N+ |** P(n) == false)}
  + Do a proof of contradiction assuming that C is non-empty
  + Using the well-ordering principle: take the least element *x* → prove that P(x) = true, or that there is some element of C < x
  + Conclude that C must be empty → no counterexamples exist → P(t) is true for all non-negative **Z**

**Chapter 6** - counting

* Product rule - for a procedure that takes n tasks to complete: the total number of ways to complete the procedure = a1 \* a2\* a3\* … \*an where ai = the number of ways to complete the nth task
* Sum rule - if a task can be done in n1 ways OR n2 ways, then the total number of ways to complete that task = n1 + n2 
  + Important: the ways to do a task represented by n1 and n2 must be disjointed (no overlap)
* Subtraction rule (inclusion - exclusion) - if a task can be done in n1 ways OR n2 ways, then the total number of ways to complete a task = n1 + n2 - (n1 AND n2)
  + Basically sum rule = subtraction rule when (n1 AND n2) = 0
* Division Rule - if a task can be done in n ways, but those n ways can be broken up into 4 groups, then the total # of ways to do a task = n/d
* R-permutation P(n,r) : r = the # of spots to fill n = total # of members | P(n,r) = n! / (n-r)!
  + Applicable to when order matters when selecting
* R-combination C(n,r) = n! / r!(n-r)!
  + Applicable to an unordered selection of objection
  + R-combination = binomial coefficient
* Combinatorial proof - use counting arguments to prove that both sides of identity count the same objects just in different ways(double counting proof) or that there is a bijection between 2 sets of objects(bijective proof)
  + One easy way of doing that = C(n,r) = a bit string of n length, and you are trying to place r 1’s
* **Binomial Theorem**
* **Quadratic Formula** to find the roots for a polynomial

Important corollaries:

* n and r = non-negative integers where r <= n, C(n,r) = C(n,n-r)
* Pascal’s identity = C(n+1,k) = C(n,k-1) + C(n,k) (try exercise 6.4.23)
* Generalized pascal (from hw 6) = 
* Vandermonde’s identity = C(m+n , r) =
* C(2n,n) =
* C(n+1,r+1) =

Different scenarios → type of calculation

* **Ordering r elements out of n with no replacement →** P(n,r)
* **Unordered r elements out of n with no replacement** → C(n,r)
* **Ordered r elements out of n with replacement** → nr
* **Unordered r elements out of n with replacement** → C(n+r-1,r) = C(n+r-1 , n-1)
  + Problems like this can be represented with trying to fill n-1 + r slots with bars and stars
  + There are n-1 bars that represent the type of thing you are picking from
  + There are r stars that represent actually selecting a type
* **Ordered r elements with k types of indistinguishable n objects →**  where nk = the number of objects that are indistinguishable from each other
  + Example on pg 450
* Unlabeled vs labeled boxes/objects:
* **Labelled objs. Into labeled boxes** = similar to ordered elements with indistinguishable objects: where ni= # of objects you put into a box
* **Unlabelled objs. Into labeled boxes** = C(n -1 + r,r) = this is just a combination with replacement
* **Labeled objects into unlabeled boxes** = S(n,j) = *this equation is going to be given to use*
* **Unlabeled objects into unlabeled boxes** = just enumerate, not given a formula: has something to do with division rule

Pigeonhole principle: if you have k boxes and k+1 objects to put into those boxes → than at least 1 box will have at least 2 or more objects in it

* Basic corollary = function from set A (|A| = k+1) to set B (|B| = k) can not be one-to-one
* Generalized principle = k boxes, N objects: at least one box will contain at least ceiling(N/k) objects

**Chapter 9** - Relations / Sets

* Properties of a relation:
  + Reflexive on a set: for every element *a* in set A, the pair (a,a) ∈ R
  + Symmetric on a set: (a,b) ∈ R → (b,a) ∈ R is true
  + Antisymmetric on a set: if (a,b) ∈ R AND (b,a) ∈ R then b=a,
    - Basically, both pairs will only both be present if b=a
  + *Symmetric and antisymmetric aren’t mutually exclusive: the empty set, or a set made up of pairs where a=b are both*
  + Transitive on a set: if (a,b) ∈ R AND (b,c) ∈ R → (a,c) ∈ R
* Relations between sets are all subsets of A X B *(all the possible combinations of (a,b) when a ∈ A and b ∈ B)*
* Relations can be combined in the same ways as Sets (using the same operators + same results)
* Composite relations = S ◯ R where R is a relations from A to B, and S is a relations from B to C and where each element of the composite (a,c) has some element b ∈ B, where (a,b) ∈ R AND (b,c) ∈ S
  + An example of this type of relation = powers relation: R = relation on set A → Rn for n = 1,2,3,4 is defined recursively
  + R1 = R
  + Rn+1 = Rn ◯ R
* **N-ary relations** = given set A1, A2, A3, … An → an n-rary relation on these sets = subset of A1 X A2 X A3… X An
  + Domain = the sets that comprise the N-ary relation
  + Degree = the number of sets in the domain
* Representing relations:Using 0-1 matrix
  + Given 2 sets A = {a1, a2, a3 .. am} and set B = {b1,b2,b3,...bn} , then in the matrix: follow this logic:
  + For a member of the matrixmij = 1 if (ai, bj) ∈ R | 0 if (ai, bj) ∉ R.
* The matrix representation can be used to represent the state of relational operators
  + Union of relations = **M**R1 V **M**R2 : if there is a 1 at position (i,j) in either matrix → (i,j) in union = 1
  + Intersection of relations = **MR1** Λ **M**R2 : if there is 1 at position (i,j) in both matrices → (i,j) in interaction = 1, else 0
  + Composite of relations (S ◯ R) = **M**R ⊙**M**S : boolean product of both matrices
  + The powers of a relation = basically an extension of composite of a relation
* Representing relations on a set A using digraph:
  + The elements of A are represented as vertices, ordered pairs are represented with edges between elements (initial vertex, terminal vertex)
* Relation Closure - closure of R, a relation on A, in relation to **P** (some relational property like reflexive / symmetric) = the relation S, another relation on A, that contains R with property **P** and is a subset of A X A that contains R AND has property **P** (the smallest possible set)
  + If relation R doesn’t have some property **P** → what is the smallest relation S that contains R but also has property **P**

* For a relation on a set A, there exists a path of length n (where n ∈ **Z+**) from a to b iff (a,b) ∈ Rn (the nth-power of R)
* Connectivity relation(R\*) - if R = relation on set A, R\* = pairs (a,b) where there is a path of at least 1 from a to b in R
  + Every element in A is connected in R\*
  + This represents the union of all Rn
  + The transitive closure of R = R\*
* The matrix representation of R\* = **M**R V **M[2]**R V **M[3]**R V … V **M[n]**R
* Equivalence relation - relation on set is reflexive, symmetric, and transitive
  + If elements are related in an equivalence relation they are equivalent (a ~ b) {pair (a,b) = a and b are related}
* Equivalence class - for a given equivalence relation, all elements related to element a are part of the equivalence class of a ⇒ [a]R
  + Classes of 2 elements in set A are either identical or disjoint (partitions)
  + Partition = nonempty subset of S that when you union all partitions = S
* Partial ordering - a relation on set S, that is reflexive, antisymmetric, and transitive
  + Poset = the combination of a set S and its partial ordered relation R → (S,R)
  + Elements a,b of a poset (S, ≼ ) are comparable if a ≼ b and b ≼ a (if a and b are related in the relation)
  + When every element in a partial ordering is comparable → partial ordering = total ordering (or chain)
* Well-ordered set - a poset where ≼ is a total ordering + every nonempty subset of S has a least element
  + Only applicable with non-negative sets because negative sets have no least element
* Well-ordered induction: trying to prove that P(x) is true for all x ∈ S where S is a well-ordered set
  + For every y ∈ S, if P(x) is true for all x ∈ S where x < y, then P(y) is true
* Hasse Diagrams - starting with a digraph representing some partial ordering
  + 1) remove all reflexive loops (a,a) edges
  + 2) remove all transitive edges
  + 3) position the vertices so that the initial vertex of an edge is lower than the terminal one
  + The resultant diagraph is called a covering relation
* Maximal vs minimal elements: the top and bottom of a hasse diagram
  + There can be multiple maximal and minimal elements
  + All it means is that in the Hasse diagram, those elements either have only incoming edges(maximal) or outgoing edges(minimal)
* Least and greatest must be unique unlike maximal or minimal
* Upper bound vs lower bound - given some subset A of the poset (S, ≼): for all a ∈ A, there are elements where a ≼ u (upper) or l ≼ a(lower), these are your bounds
  + Least upper bound is a unique upper bound that is less than all other upper bounds
  + Greatest lower bound is the analogous concept
* Lattice = when all pairs of elements have a least upper bound and a greatest lower bound

**Chapter 10** - Graphs

* Graphs that may include loops, and possibly multiple edges connecting the same pair of vertices or a vertex to itself, are sometimes called **pseudographs**.
* The degree of a vertex in an undirected graph is the number of edges incident with it, except that **a loop at a vertex contributes twice to the degree of that vertex.** The degree of the vertex v is denoted by deg(v).
* Theorem 1: The handshaking theorem. Let G = (V, E) be an undirected graph with m edges. Then 2m = Σdeg(v) (v ∈ V) (Note that this applies even if multiple edges and loops are present.)
* Theorem 2: An undirected graph has an even number of vertices of odd degree.
* The vertex u is called the initial vertex of (u, v), and v is called the terminal or end vertex of (u,v).
* **Some Special Simple Graphs**
  + Complete Graphs Kn

A complete graph on n vertices, denoted by **Kn**, is a simple graph that contains exactly one edge between **each pair of distinct vertices**. A simple graph for which there is at least one pair of distinct vertex not connected by an edge is called **noncomplete**.

* + Cycles Cn

A cycle **Cn**, n ≥ 3, consists of n vertices v1, v2, ... , vn and edges {v1, v2}, {v2, v3}, ... , {vn−1, vn}, and {vn, v1}.

* + Wheels Wn

Add an additional vertex to a cycle Cn, for n ≥ 3, and connect this new vertex to each of the n vertices in Cn, by new edges.

* + n-Cubes Qn

A graph that has vertices representing the 2^n bit strings of length n. Two vertices are **adjacent** if and only if the bit strings that they represent **differ in exactly one bit position**.

* **Bipartite Graphs**

Its vertex set V can be partitioned into two **disjoint** sets V1 and V2 such that every edge in the graph connects a vertex in V1 and a vertex in V2 (so that no edge in G connects either two vertices in V1 or two vertices in V2).

* **Complete Bipartite Graphs**

*Km,n* is a graph that has its vertex set partitioned into two subsets of *m* and *n* vertices, respectively with an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset.

* **Subgraph & Proper Subgraph**

A subgraph of a graph G = (V, E) is a graph H = (W, F), where W ⊆ V and F ⊆ E. A sub- graph H of G is a proper subgraph of G if H ≠ G.

* Removing and Adding edges of a graph:
  + Removing: G - e = (V, E - {e})
  + Adding: G + e = (V, E U {e})
* Edge Contractions
  + Definition: removes an edge *e* with endpoints *u* and *v* and merges *u* and *w* into a new single vertex *w*, and for each edge with *u* or *v* as an endpoint replaces the edge with one with *w* as endpoint in place of *u* or *v* and with the same second endpoint.
  + G = (V, E) ⇒ G’ = (V’, E’) , V’ = V - {u,v} U {w},
* Removing Vertices from a graph: remove a vertex v and all edges incident to it from G = (V,E)
* G1 U G2 = (V1 U V2, E1U E2)
* Representing Graphs :
  + adjacency lists
  + adjacency matrix
  + Incidence matrices:
    - row -vertices
    - column -edges
* Isomorphism of Graphs
  + Definition: The simple graphs G1 = (V1, E1) and G2 = (V2, E2) are isomorphic if there exists a one-to-one and onto function f from V1 to V2 with the property that a and b are adjacent in G1 if and only if f(a) and f(b) are adjacent in G2, for all a and b in V1. Such a function f is called an isomorphism. Two simple graphs that are not isomorphic are called nonisomorphic.
  + How to check?
    - number of vertices and number of edges
    - Number of vertices with same degree
    - Find corresponding vertices in G2 to the vertices in G1, for example the image of u1 in G1 is either v4 or v6 in G2.
      * Arbitrarily set f(u1) = v6 [If we found that this choice did not lead to isomorphism, we would then try f(u1) = v4]
      * Find the adjacent vertice to u1 and v6
    - Examine the adjacency matrix of G1 and G2. If they are the same, we conclude that f is an isomorphism, then G1 and G2 are isomorphic. If not, we need to check another assumption, such as f(u2) = v4.
* Connectivity
  + A path of length n from u to v in G is a sequence of n edges e1, …, en of G. The path or circuit is said to pass through the vertices x1, x2, …, xn-1 or traverse the edges e1, e2, …, en. A path or circuit is simple if it does not contain the same edge more than once.
  + A path of length greater than zero that begins and ends at the same vertex is called a circuit or cycle. A path or circuit is called simple if it does not contain the same edge more than once.
  + There is a simple path between every pair of distinct vertices of a connected undirected graph.
  + A connected component of a graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G. That is, a connected component of a graph G is a maximal connected subgraph of G.
  + Vertex connectivity
    - Not all graphs have cut vertices, such as the complete graph Kn. Connected graphs without cut vertices are called non-separable graphs, and can be thought of as more connected than those with a cut vertex.
    - A subset V’ of the vertex set V of G = (V,E) is a vertex cut, or separating set, if G - V’ is disconnected.
    - Every connected graph, except a complete graph, has a vertex cut.
    - The vertex connectivity of a noncomplete graph G, denoted by k(G), as the minimum number of vertices in a vertex cut. Set k(Kn) = n - 1, the number of vertices needed to be removed to produce a graph with a single vertex, which means a complete graph needs to remove n-1 vertices to leave only one vertex to be disconnected.
    - The larger k(G) is, the more connected we consider G to be. k(K1) = 0
    - A graph is k-connected (or k-vertex-connected), if k(G) ≥ k
      * A graph G is 1-connected if it is connected and not a graph containing a single vertex.
      * A graph is 2-connected, or biconnected, if it is nonseparable and has at least three vertices.
      * If G is a k-connected graph, then G is a j-connected graph for all j with 0 ≤ j ≤ k. (?)
  + Edge Connectivity
    - Definition: the edge connectivity of a graph G, denoted by 𝜆(G), is the minimum number of edges in an edge cut of G.
    - 𝜆(G) = 0 if G is not connected or a graph consisting of a single vertex.
    - 0 ≤ 𝜆(G) ≤ n-1 if G is a graph with n vertices. 𝜆(G) = n-1 if and only if G = Kn. So 𝜆(G) ≤ n-2 when G is not a complete graph.
  + Isomorphic (P723)
    - Same number of vertices
    - Same number of edges
    - Same degrees of vertices
    - Same circuit of length k, where k is a positive integer greater than 2.
    - If it is hard to determine, we can use mapping (P723, Example 14 )
* Euler and Hamilton Paths
  + Euler Paths and Circuit
    - Euler Paths:
      * Definition: a simple path containing every edge of G
      * Start from one vertex and end at another vertex
    - Euler Circuit:
      * Definition: a simple circuit containing every edge of G
      * Start and end at the same vertex
  + Hamilton Paths and Circuits
    - Hamilton Path
      * Definition: a simple path in a graph G that passes through every vertex exactly once is called a Hamilton path.
      * Start from one vertex and end at another vertex
    - Hamilton Circuit
      * Definition: a simple circuit in a graph G that passes through every vertex exactly once is called a Hamilton circuit.
      * Start and end at the same vertex
  + Planar Graphs
    - Definition: A graph is called planar if it can be drawn in the plane without any edges crossing (where a crossing of edges is the intersection of the lines or arcs representing them at a point other than their common endpoint). Such a drawing is called a planar representation of the graph.
    - Euler’s Formula:
      * Definition: Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G. Then r = e - v + 2.
      * Proof: use inductive definition, two cases
        + In the first case, both ak+1 and bk+1 are already in Gk.
        + In the second case, one of the two vertices of the new edge is not already in Gk.

**Terms**

Relation - a binary relation from A to B = a subset of A X B

N-ary relation - relation on sets A1, A2, A3 …, An: the n-ary relation is a subset of A1 X A2 X A3 … An

Function - function from A to B = for every element of A, there is exactly 1 mapping to B

Surjective(onto) - every member of the codomain is mapped to only at most once by elements of the domain

injective(one-to-one) - every member of the codomain is mapped to by the domain

bijective(one-to-one correspondence) - both onto and one-to-one

Cartesian product - the set of ordered pairs (a,b) where a ∋ A and b ∋ B

Set - unordered collection of **distinct** objects

<lots of question from chapter 9 + chapter 10 (graph)>

<will take a question from quiz 1 / quiz 2 and put it in final>