CS 520 Algorithm Analysis
Spring 2017
Lecture Week 11

I. Chapter 34 NP-Completeness
  A. Introduction
    1. Turing Machines (Chapter 13, Discrete Structures, Logic, and Computability, James L. Hein)
      a) Turing Machine 1936
        (1) Person doing primitive calculation on a long strip of paper divided into contiguous individual cells
          (a) Cells can each hold a symbol from a fixed alphabet
        (2) Starting at some cell, observe the symbol in the cell
          (a) Leave it alone
          (b) Erase it, and replace it with another symbol
          (c) Move to another adjacent cell
      b) Model abstraction
        (1) Tape and control unit
        (2) Instructions
          (a) Current state of the machine
          (b) A tape symbol read from the current tape cell
          (c) A tape symbol to write into the current tape cell
          (d) A direction for the tape head to move
          (e) The next machine state
          (f) Interpretation <*i, a, b, L, j*>
            (i) If the current state of the machine is *i*, and if the symbol in the current tape cell is *a*, then write *b* into the current tape cell, move one cell to the left, and go to state *j*
        (3) An input string is represented on the tape by placing the letters of the string in contiguous tape cells; all other cells contain the symbol representing “blank”
        (4) The tape head is positioned at the leftmost cell of the input string unless specified otherwise
        (5) There is one start state
        (6) There is one halt state
      c) An input string is accepted by a Turing machine if the machine enters the halt state, otherwise the input string is rejected
        (1) Reject if stops on a state other from the halt state from which there is no move
        (2) Reject if machine runs for ever
      d) Construct a machine to accept the language $\{a^n b^m \mid m,n \in N\}$
    2. Turing's Halting Problem that cannot be solved by any computer, no

matter how much time is provided
   a) https://www.cs.auckland.ac.nz/~chaitin/unknowable/
   b) https://www.cs.auckland.ac.nz/~chaitin/sciamer3.pdf .
   c) Some example problems that are unsolvable for any arbitrary Turing Machine *M* or any other equivalent computational model
      (1) Does *M* halt when started on the empty tape
      (2) Is there an input string for which *M* halts
      (3) Does *M* halt on every input string?

3. Section 13.2 The Church-Turing Thesis (Discrete Structures, Logic, and Computability, James L. Hein)
   a) A computation is the execution of an algorithm
   b) "Computable" has something to do with a formal process (execution) and a formal description (algorithm)
   c) Examples
      (1) The derivation processes associated with grammars
      (2) The evaluation process associated with functions
      (3) The state transition process associated with machines
      (4) The execution process associated with programs and programming languages
   d) Modeling the idea of computability
      (1) Is one model more powerful than another? That is does one model solve all the problems of another model and also solve some problem that is not solvable by the other model?
      (2) Is there a most powerful model?
   e) Some concrete examples
      (1) Turing machines are more powerful than pushdown automata
      (2) Pushdown automata are more powerful than deterministic finite automata
   f) Church-Turing Thesis
      (1) Anything that is intuitively computable can be computed by a Turing machine
      (2) Once you know some computational model *M* is equivalent to the Turing machine model, then
         (a) Church-Turing Thesis for *M* – Anything that is intuitively computable can be computed by the *M* computational model
      (3) https://stanford.library.sydney.edu.au/entries/church-turing/
   g) Why this is important
      (1) No one has ever invented a computational model more powerful than a Turing machine

4. think of polynomial time problems are tractable, superpolynomial time algorithms as intractable
   a) http://homepage.divms.uiowa.edu/~ljay/publications.dir/Qorder.pdf
   b) *A NOTE ON Q-ORDER OF CONVERGENCE*, L.O. Jay

c) Actually the Wikipedia entry is accurate -- http://en.wikipedia.org/wiki/Rate_of_convergence .

5. NP-Complete problems - status of solution time is unknown
   a) no polynomial time algorithm has yet been discovered for an NP-complete problem
   b) no one has been able to prove a superpolynomial time bound for any either
6. P ≠ NP question
7. If any NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial time algorithm

B. Decision Problems
1. a decision problem is a question that has two possible answers "yes" and "no"
2. the question is about some input
3. a problem instance is the combination of the problem and a specific input
   a) The instance description part defines the information expected in the input
   b) The question part states the actual yes-or-no question; contains variables defined in the instance description
4. can be thought of as mapping all inputs into the set {*yes, no*}
5. example 1
   a) Instance: an undirected graph $G=(V,E)$
   b) Question: Does G contain a clique of k vertices? (a clique is a complete subgraph: every pair of vertices in the subgraph has an edge between them)
   c) Runs in $O(k^2 n^k)$
      (1) k is not part of the input so it does not vary from one instance to another
      (2) If k is regarded as a constant, then this algorithm runs in polynomial time
6. example 2
   a) Instance: an undirected graph $G=(V,E)$ and an integer k
   b) Question: Does G contain a clique of k vertices?
   c) Runs in $O(k^2 n^k)$
      (1) k is part of the input; it is a variable for each problem instance
      (2) The problem does not run in polynomial time because the exponent of n is a variable

C. Chapter 34 Introductory Information
1. can all problems be solved in polynomial time? (no)
   a) Turing's Halting Problem
2. NP-Complete problems have unknown runtime status
   a) No polynomial time algorithm has yet been discovered for problems in this set
   b) No proof that no polynomial time algorithms exist either
   c) Is P ≠NP has not been proven
3. Similarities between P and NP problems
   a) Shortest vs. longest simple paths
   b) Euler tour vs. Hamiltonian cycle
   c) 2-CNF vs 3-CNF satisfiability

4. NP-completeness and the classes P and NP (informal definitions)
    a) Class P
        (1) Consists of problems solvable in polynomial time $O(n^k)$ for a constant k (remember, k is not part of the input) and input size n
    b) Class NP
        (1) Problems that are "verifiable" in polynomial time
            (a) If given a certificate of a solution can verify that the certificate is correct in polynomial time in the size of the problem
    c) Any problem in P is also in NP, since problems in P can be solved in polynomial time even without a certificate
        (1) Is P a proper subset of NP? Is the question
    d) Class NPC
        (1) If the problem is in NP and is as "hard" as any problem in NP later in the chapter
        (2) If any NP-complete problem can be solved in polynomial time then every NP-complete problem has a polynomial time algorithm
5. Why it is important to be familiar with these problem classes

D. Overview of showing problems to be NP-complete
1. we are making a statement about how hard the problem is, not about how easy it is
2. trying to prove that no efficient algorithm is likely to exist
3. Three concepts
    a) Decision problems vs. optimization problems
        (1) NP-completeness applies directly to decision problems (answer is either yes or no)
        (2) There is a relationship between optimization and decision problems – cast the optimization problem as a decision problem by imposing a bound on the value to be optimized
        (3) If an optimization problem is easy, its related decision problem is easy as well
    b) Reductions
        (1) Suppose we have an instance of a decision problem we would like to solve in polynomial time (A)
        (2) Suppose there is a different decision problem we already know how to solve in polynomial time (B)
        (3) Suppose we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:
            (a) The transformation takes polynomial time
            (b) The answers are the same (if answer to α is "yes" then the answer to β is "yes")
        (4) Solving problem A in polynomial time
            (a) Given and instance α of problem A use a polynomial-time reduction algorithm to transform it to an instance β of B

(b) Run the polynomial-time decision algorithm for *B* on the instance β
(c) Use the answer β for the answer for α

(5) Use polynomial time reductions in the opposite way to show that a problem is NP-complete (show that no polynomial time algorithm can exist for a particular problem *B*)
(a) Suppose we have a decision problem *A* for which we already know that NO polynomial time algorithm can exist
(b) Suppose that we have a polynomial time reduction transforming instances of *A* to instances of *B*
(c) Use proof by contradiction to show that no polynomial time algorithm can exist for *B*

c) A first NP-complete problem
(1) Use circuit satisfiability as the base problem for NP-completeness reduction algorithm proofs

E. 34.1 Polynomial Time

1. Three supporting arguments for why, philosophically, polynomial time algorithms are generally regarded as tractable
a) although it is reasonable to regard a problem that requires time $\Theta(n^{100})$ as intractable, there are very few practical problems that require time on the order of such high-degree polynomials
b) a problem that can be solved in polynomial time in one model can be solved in polynomial time in another (serial random-access machine and Turing machine models)
c) class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication and composition
2. ***See Tarjan notes on rooted tree formulation***
3. Abstract Problems
a) abstract problem *Q* is the binary relation of set *I* of problem instances and a set *S* of problem solutions
b) NP-completeness restricts attention to decision problems (yes/no solutions)
(1) A decision problem is a question that has two possible answers (*yes* or *no*)
(2) The question has some *input*
(3) A *problem instance* is the combination of the problem and a specific input
(4) Usually the statement of the decision problem has two parts
(a) The *instance description* defines the information expected in the input
(b) The *question* part states the actual yes-or-no question; it contains variables defined in the instance description
(5) Examples
(a) *Instance*: an undirected graph *G* = (*V, E)*

(b) *Question*: Does *G* contain a clique of *k* vertices (a *clique* is a complete subgraph: every pair of vertices in the subgraph has an edge between them)?
(c) *Instance*: an undirected graph $G = (V, E)$ and integer *k*
(d) *Question*: Does *G* contain a clique of *k* vertices?.

c) many abstract problems are not decision problems, but rather optimization problems in which some variable must be minimized or maximized. Recast as an NP-completeness problem by imposing a bound on the value to be optimized. If the decision problem is easy to solve, the optimization problem is easy to solve (this is where the algorithm researchers I personally know exert most of their efforts)

d) Sample Problems (Chapter 13 Baase and Van Gelder)

(1) Graph coloring and chromatic number

(a) A *coloring* of a graph $G = (V, E)$ is a mapping $C: V \rightarrow S$, where *S* is a finite set (of colors"), such that if $vw \in E$ then $C(v) \neq C(w)$; in other words adjacent vertices are not assigned the same color. The *chromatic number* of *G*, denoted $\chi(G)$, is the smallest number of colors needed to color *G* (that is the smallest *k* such that there exists a coloring *C* for *G* and $|C(V)| = k$.
(b) *Optimization problem:* Given *G*, determine $\chi(G)$ (and produce an optimal coloring, that is, one that uses only $\chi(G)$ colors)
(c) *Decision Problem*: Given *G* and a positive integer *k*, is there a coloring of *G* using at most *k* colors? (if so, *G* is said to be *k*-colorable)

(2) Subset Sum

(a) The input is a positive integer *C* and *n* objects whose sizes are $s_1, s_2, \ldots, s_n$.
(b) *Optimization problem:* Among subsets of the objects with sum at most *C,* what is the largest subset sum?
(c) *Decision Problem*: Is there a subset of the objects whose sizes add up exactly to *C?.*

4. Encodings

a) an encoding of a set *S* of abstract objects is a mapping *e* from *S* to the set of binary strings
b) a computer algorithm that "solves" some abstract decision problem actually takes an encoding of the problem instance as input
c) a concrete problem has instance set as the set of binary strings

d) an algorithm solves a concrete problem in O(T(*n*)) if when provided a problem instance *i* of length *n*= | *i* | , the algorithm can produce a solution in at most O(T(*n*)) time
e) complexity class P as the set of concrete decision problems that are solvable in polynomial time
f) the efficiency of solving a problem should not depend on how the problem is encoded (but it really does depend on the encoding)
    (1) unary input versus binary input example on page 1056
g) want to rule out expensive encodings; the actual encoding starts to make less difference
h) polynomial-time computable and polynomially related definitions on page 1056

5. A formal-language framework
a) the set of instances for any decision problem *Q* is the set Σ^*, where $\Sigma = \{0,1\}$. $L = \{x \in \Sigma^* : Q(x) = 1\}$
b) PATH example page 1058
c) algorithm A accepts a string $x \in \{0,1\}^*$ if, given input *x*, the algorithm outputs A(*x*) =1; rejects a string otherwise
d) A language L is decided by an algorithm A if every binary string is either accepted or rejected by the algorithm
    (1) accepted in polynomial time
    (2) decided in polynomial time
e) a complexity class is a set of languages, membership in which is determined by a complexity measure, such as running time, on an algorithm that determines whether a given string x belongs to language *L*
f) The class P
    (1) Def. 13.2 pg 553 (Baase and Van Gelder) Polynomially bounded
        (a) An algorithm is said to be polynomially bounded if its worst-case complexity is bounded by a polynomial function of the input size (i.e., if there is a polynomial *p* such that for each input of size *n* the algorithm terminates after at most *p(n)* steps)
    (2) Def. 13.3 pg 553 (Baase and Van Gelder) The class P
        (a) P is the class of decision problems that are polynomially bounded
    (3) Our text, page 1059
        (a) Complexity class P as the set of concrete decision problems that are solvable in polynomial time
    (4) Our text, page 1059
        (a) P = {$L \subseteq \{0,1\}^*$: there exists an algorithm *A* that decides *L* in polynomial time}
        (b) P = {*L* : *L* is accepted by a polynomial-time

algorithm}.

F. 34.2 Polynomial-time verification

1. Hamiltonian cycles - cycle that include each vertex in the graph in the path, but only once per vertex
2. verification algorithms
   a) defined as a two-argument algorithm A
      (1) one argument is an ordinary input string *x*
      (2) other argument is a binary string *y* called a certificate
   b) A verifies an input string *x* if there exists a certificate *y* such that A(*x,y*) = 1
   c) *L* = {x $\in$ {0,1}* : there exists *y* $\in$ {0,1}* such that *A*(*x,y*) = 1}.
3. ***Skip below to Tarjan and Horowitz & Sahni for discussion of non-determinstic algorithms***
4. the complexity class NP
   a) complexity class NP is the class of languages that can be verified by a polynomial time algorithm
   b) the class of decision problems for which a given proposed solution for a given input can be checked quickly (in polynomial time) to see of it is really a solution (i.e., if it satisfies all the requirements of the problem (Baase and Van Gelder page 554)
   c) a language *L* belongs to NP IFF there exists a two-input algorithm *A* and constant *c* such that *L* = {x $\in$ {0,1}* : there exists *y* with |*y*| = *O*(|*x*|^c) such that *A*(*x,y*) = 1}.
   d) Definition 13.5 The Class NP (Baase and Van Gelder page 557)
      (1) NP is the class of decision problems for which there is a polynomially bounded nondeterministic algorithm.
   e) P $\subseteq$NP
      (1) P consists of problems that can be solved quickly
      (2) NP consists of problems that can be verified quickly
      (3) There has not been a single problem in NP for which it has been proved that the problem is not in P
      (4) There are no polynomially bounded algorithms known for many problems in NP, but no larger-than-polynomially lower bounds have been proved with these problems.
   f) complexity class co-NP as the set of languages *L* such that *L*-complement $\in$ NP; the question of whether NP is closed under complement has not been resolved
   g) Is nondeterminism more powerful than determinism in the sense that some problems can be solved in polynomial time with a nondeterministic "guesser" that cannot be solved in polynomial time by an ordinary algorithm?
      (1) If a problem is in NP, with polynomial time bound, say *p*, we can deterministically give the proper

answer (yes or no) if we check all strings of length at most $p(n)$.

(2) The number of steps needed to check each string is at most $p(n)$

(3) Trouble is that there are too many strings to check. If there are c characters in the character set, then there are $c^{p(n)}$ strings of length $p(n)$ – the number of strings is exponential in n

G. The size of the Input (Section 13.2.5 Basse and Van Gelder)

1. Problem – given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (Is n non-prime?)
2. Is the problem in P?
3. consider algorithm

```
factor = 0;
for (j=2; j < n; j++)
   if ((n mod j) == 0)
      factor=j;
      break;
return factor;
```

4. runtime analysis
   a) loop body executed fewer than n times
   b) (n mod j) evaluated in $O(\log^2(n))$
   c) running time $O(n^2)$
5. Problem of determining whether an integer is prime or factorable is not known to be in P (that is why it is the basis for encryption)
6. How to resolve this apparent paradox?
   a) The input is n, but what is the size of n?
   b) The size of an input is the number of characters it takes to write the input
   c) Decimal notation is roughly $\log_{10} n$, $\lg n$ in binary notation
   d) If the input size s is $\log_{10} n$, and the running time of the algorithm is n, then the running time is an exponential function of the input size ($n=10^s$), so the algorithm for determining if n is prime is not in P.
   e) Question "Is integer n prime?" is in NP

H. 34.3 NP-Completeness and reducibility

1. Reducibility
   a) intuitively a problem Q can be reduced to a problem Q' if any instance of Q can be "easily paraphrased" as an instance of Q', the solution to which provides a solution to the instance of Q
   b) L_1 is polynomial-time reducible to a language L_2 if there exists a polynomial-time computable function $f:\{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $z \in \{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$
   c) Definition 13.6 (Baase and Van Gelder pp 560-561) polynomial reduction and reducibility (restated in terms of our text)
      (1) Let F be a function from the input set for a decision problem L_1 into the set for a decision problem L_2. F is a *polynomial reduction* (also called a *polynomial transformation*) from L_1 to L_2

if all of the following hold:
(2) F can be computed in polynomially bounded time
(3) For every string x, if x is a *yes* input for L_1, then $F(x)$ is a *yes* for L_2
(4) For every string x, if x is a *no* input for L_1, then $F(x)$ is a *no* for L_2
(a) It is usually easier to prove the contrapositive of part (4)
(b) For every string x, if $F(x)$ is a *yes* input for L_2, then x is a *yes* for L_1
(5) Problem L_1 is *polynomially reducible* to L_2 if there exists a polynomial transformation from L_1 to L_2.

d) Lemma 34.3 our text for proving that a language belongs to class P

2. A reduction example
a) Let the problem **P** be: Given a sequence of Boolean values, does at least one of them have the value *true*? (in other words, this is a decision problem version of computing the *n*-way Boolean *or*, when the string has *n* values)
(1) Let **Q** be: given a sequence of integers, is the maximum of the integers positive?
(2) Let the transformation T be defined:
(a) $T(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$
(b) Where $y_i = 1$ if $x_i =$ *true*, $y_i = 0$ if $x_i =$ *false*
b) An algorithm to solve **Q** when applied to the y's solves **P** for the x's

3. NP-Completeness
a) Describes decision problems that are the hardest ones in NP in the sense that, if there were a polynomially bounded algorithm for an NP-complete problem, then there would be a polynomially bounded algorithm for each problem in NP
(1) A problem L_2 is NP-hard if every problem L_1 in NP is reducible to L_2: that is $L_1 \leq_P L_2$.
(2) A problem L_2 is NP-complete if it is in NP and is NP-hard
b) A language $L \subseteq \{0,1\}^*$ is NP-complete if
(1) $L \in$ NP and
(2) L' is polynomial-time reducible to L for every $L \in$ NP
c) if a language L satisfies property (2) but not necessarily property (1) we say that L is NP-hard

4. circuit satisfiability
a) once we prove at least one problem is NP-complete, we can use polynomial-time reducibility as a tool to prove the NP-completeness of other problems
b) a truth assignment for a Boolean combination circuit is a set of Boolean input values
c) a one-output Boolean combinatorial circuit is satisfiable if it has a satisfying assignment: a truth assignment that causes the output of the circuit to be 1

d) "Given a Boolean combinatorial circuit composed of AND, OR, and NOT gates, is it satisfiable?"
e) The circuit-satisfiability problem belongs to the class NP: can verify in polynomial time
f) the circuit-satisfiability problem is NP-hard
g) the circuit-satisfiability problem is NP-complete

I. 34.4 NP-completeness proofs
1. page 1078 method for proving a language L is NP-complete
a) relies on Lemma 34.8 and steps following.
2. proving a problem is NP-complete (Baase and Van Gelder page 563)
a) to show that the problem L_2 is NP-complete, choose some known NP-complete problem L_1 and reduce L_1 to L_2, not the other way around
(1) Since L_1 is NP-complete, all problems $R \in$ NP are reducible to L_1; that is $R \leq_P L_1$.
(2) Show that $L_1 \leq_P L_2$.
(3) Then all problems $R \in$ NP satisfy $R \leq_P L_2$ by transitivity of reductions
(4) Therefore L_2 is NP-complete.
3. formula satisfiability: map a Boolean circuit to a Boolean formula
4. 3-CNF satisfiability
a) a literal in a Boolean formula is an occurrence of a variable or its negation
b) a Boolean formula is in conjunctive normal form (CNF) if it is expressed as an AND of clauses, each of which is the OR of one or more literals
c) in 3-CNF, each clause has exactly 3 distinct literals
5. disjunctive normal form - an OR of ANDs

J. 34.5 NP-complete Problems
1. 34.5.1 the clique problem
a) a clique in an undirected graph is a subset of vertices, each pair of which is connected by an edge in E; a complete subgraph of G
b) size of a clique is the number of vertices it contains
c) clique problem is the optimization problem of finding a clique of maximum size in a graph
2. 34.5.2 the vertex cover problem
a) a vertex cover of an undirected graph is a subset of vertices such that if $(u,v) \in E$ then $u \in V'$ or $v \in V'$ (or both); each vertex covers its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E
b) the size of the cover is the number of vertices it contains
c) vertex-cover problem is to find a vertex cover of minimum size in a given graph
3. 34.5.3 the Hamiltonian-cycle problem
4. 34.5.4 the traveling salesman problem
5. 34.5.5 the subset sum problem
a) finite set $S \subset$ N and a target $t \in$ N. Is there a subset of S whose elements sum to t?

ADDITIONAL NOTES FOR CLASS

II. problem classification Tarjan(4-7) HS(501-513)

- A. Why is polynomial time so important
  1. ask for polynomial examples
  2. ask for non-polynomial examples
- B. points of interest
  1. to solve a problem means to solve all instances of the problem
  2. examine algorithms with worst possible input data
  3. interested in large problems
  4. the choice of a parameter to represent the size of a problem will not alter the classification of a problem
  5. the encoding of input will not change the problem's classification
  6. classification is independent of machine model used
- C. we will use the following classifications
  1. undecidable problems (unsolvable problems): problems for which no algorithm can be written (Turing machine halting problem)
  2. intractable problems (provably difficult problems): problems for which no polynomial algorithm can possibly be developed to solve (only exponential algorithms are expected to solve these problems)
  3. NP problems: solvable in polynomial time if we can guess correctly which computational path to follow (includes problems that have exponential algorithms but which have not been proved that they cannot have polynomial-time algorithms
  4. P problems: all problems that have polynomial time algorithms
- D. Venn diagram
  1. NP contains all
  2. P is a subset of NP
  3. NP-complete is a subset of NP
- E. an efficient algorithm is one which is O(polynomial f(n))
- F. tractable problems have efficient solutions, intractable problems do not
- G. NP-hard: if an NP-hard problem can be solved in polynomial time then all NP-complete problems can be solved in polynomial time
- H. NP-complete: can be solved in polynomial time iff all other NP-complete problems can be solved in polynomial time
- I. deterministic algorithms - property that the result of every operation is uniquely defined
- **J. non-deterministic algorithms**
  1. contain operations whose outcome is not uniquely defined but is limited to a specified set of possibilities. the machine using the algorithm is allowed to choose any one of these outcomes subject to a termination condition
     a) a non-deterministic algorithm terminates unsuccessfully iff there exists no set of choices leading to a successful signal
- K. n is the length of input to an algorithm and is used to measure complexity
  1. assume inputs are integers
     a) binary form of input (radix = 2)
     b) unary form of input (radix = 1)
- L. remember that there are exponential algorithms that are faster than polynomial time algorithms in average cases; this is why we need worst case analysis

M. problem representation

1. if a problem has answer of "yes" and "no" we can build a rooted tree where each node represents a computation and the leaves are answered "yes" or "no"
2. for an algorithm to have polynomial time, the height of the tree for any parameter n must be a polynomial of n and the amount of time needed by each node must be polynomial
3. if the "yes" answer can be verified in polynomial time, we can consider the problem can be solved by a nondeterministic algorithm in polynomial time or the problem belongs to class NP; the problem of answering "no" probably does not belong to NP for NP problems
4. when a problem is in P its complement belongs to P (not true for NP problems)

N. classes of NP-hard and NP-complete

1. P is the set of all decision problems solvable by a deterministic algorithm in polynomial time
2. NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time

O. example problems pages 545-547

III. Tarjan, 1.4 Algorithmic Notation

A. Dijkstra's guarded command language(e.w. Dijkstra, *A discipline for Programming,* Prentice-Hall, Englewood Cliffs, NJ 1976. and *Guarded Commands, Nondeterminancy and Formal Derivation of Programs*, Communications of the ACM, August 1975, Vol 18, No 8, pp 453-457.)

1. **if** statement

   if $condition_1 \rightarrow statement\ list_1$ | $condition_2 \rightarrow statement\ list_2$ | …
   | $condition_n \rightarrow statement\ list_n$ **fi**

   a) cause the conditions to be evaluated and the statement list for the first true condition to be executed
   b) if none of the conditions is true none of the statement lists is executed
2. conditional expressions

   if $condition_1 \rightarrow exp_1$ | $condition_2 \rightarrow exp_2$ | …
   | $condition_n \rightarrow exp_n$ **fi**
3. **do** statement

   do $condition_1 \rightarrow statement\ list_1$ | $condition_2 \rightarrow statement\ list_2$ | …
   | $condition_n \rightarrow statement\ list_n$ **od**

   a) cause the conditions to be evaluated and the statement list for the first true condition to be executed; if none of the conditions is true none of the statement lists is executed
   b) after the execution of the statement list, the conditions are reevaluated, the appropriate statement list is executed, and this is repeated until all conditions evaluate to false
4. **for** statement

   for *iterator* $\rightarrow$ *statement list* **rof**

   a) causes the statement list to be evaluated once for each value of the iterator
   b) an iterator has the form $x \in s$, where x is a variable and s is

an interval, arithmetic progression, list or set

c) the statement list is executed |s| times

IV. Horowitz & Sahni, Fundamentals of Computer Algorithms, Chapter 11, NP-Hard and NP-Complete Problems

A. Nondeterministic Algorithms

1. Algorithms that contain operations whose outcome is not uniquely defined but is limited to a specified set of possibilities with the machine executing allowed to choose any one of these outcomes subject to a termination condition leads to the concept of a nondeterministic algorithm
2. (Baase and Van Gelder page 555) Definition 13.4
   a) a nondeterministic algorithm has two phases and an output step
      (1) the nondeterministic "guessing" phase. Some completely arbitrary string of characters, *s*, is written beginning at some designated place in memory. Each time the algorithm is run, the string may differ (the string is a certificate; it may be thought of as a guess at a solution for the problem)
      (2) the deterministic "verifying" phase. A deterministic (i.e., ordinary) subroutine begins execution. In addition to the decision problem's input, the subroutine may use *s,* or it may ignore *s*. Eventually it returns a value *true* or *false* - or it may get in an infinite loop and never halt. (think of the verifying phase as checking *s* to see if it is a solution for the decision problem's input, i.e., if it justifies a *yes* answer for the decision problem's input)
      (3) the output step. If the verifying phase returned *true*, the algorithm outputs *yes*, otherwise, there is no output
   b) pseudocode structure
   c)
```
void nondetA(string input)
 String  s = genCertif();
 Boolean checkOK = verifyA(input, s);
 If (checkOk)
      Output "yes".
 Return;
```
3. In Horowitz and Sahni they add functions to their pseudocode
   a) **choice** (*S*) … arbitrarily chooses one of the elements of the set *S*
      (1) X ← **choice** (1:*n*) could result in *X* being assigned any integer in the range [1,*n*]
   b) **failure** … signals an unsuccessful operation
   c) **success** … signals a successful operation
4. A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal

B. Example 11.1 Consider the problem of searching for an element *x* in a given set of elements *A*(1:*n*), $n \geq 1$. We are required to determine an

index j such that $A(j) = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is

```
j ← choice (1:n)
if A(j) = x then print ( j ); success endif
print( '0' ); failure
```

1. Nondeterministic complexity $O(1)$
2. Since A is not ordered, every deterministic search algorithm is $\Omega(n)$

C. Example 11.2 [Sorting] Let $A(i)$, $1 \le i \le n$ be an unsorted set of positive integers. The nodeterministic algorithm NSORT(A, n) sorts the numbers into nondecreasing order and then outputs them in this order. Its complexity is O(n), Recall that all deterministic sorting algorithms must have a complexity of $\Omega(n \log n)$

```
procedure NSORT(A, n)
//sort n positive integers//
integer A(n), B(n), n, i, j
B ←0  //initialize B to zero
for i ←1 to n do
      j ← choice (1:n)
      if B(j) ≠ 0 then failure endif
            B(j) ← A(i)
repeat
for i ← 1 to n-1 do //verify order //
      if B(i) > B(i + 1) then failure endif
repeat
print(B)
success
end NSORT
```

D. The time required by a nondeterministic algorithm performing on any given input is the minimum number of steps needed to reach a successful completion of there exists a sequence of choices leading to such a completion

E. The Classes NP-hard and NP-complete

1. An algorithm A is of polynomial complexity if there exists a polynomial $p()$ such that the computing time of A is $O(p(n)$ for every input of size n
2. P is the set of all decision problems solvable by a deterministic algorithm in polynomial time
3. NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time
4. P $\subseteq$ NP
5. We do not know if P = NP of if P $\neq$ NP
6. A problem is NP-hard if and only if satisfiability reduces to L
7. A problem is NP-complete if and only if L is NP-hard and $L \in$ NP
8. Example 11.10 An extreme example of an NP-hard decision problem that is not NP-complete is the halting problem for deterministic algorithms
   a) The halting problem is to determine for any arbitrary deterministic algorithm A and an input i whether algorithm A with input i ever terminates
   b) It is well known that this problem is undecidable