**Introduction**

1. A problem is said to be polynomial if there exists an algorithm that solves the problem in time T(n)=O(nc), where c is a constant.
2. Examples of polynomial problems:
   * + 1. Sorting: O(n log n) = O(n2)
       2. All-pairs shortest path: O(n3)
       3. Minimum spanning tree: O(E log E)= O(E2)
3. A problem is said to be exponential if no polynomial-time algorithm can be developed for it and if we can find an algorithm that solves it in O(nu(n)), where u(n) goes to infinity as n goes to infinity.
4. The world of computation can be subdivided into three classes:
   * + - 1. Polynomial problems (P)
         2. Exponential problems (E)
         3. Intractable (non-computable) problems (I)
5. There is a very large and important class of problems that
   1. we know how to solve exponentially,
   2. we don't know how to solve polynomially, and
   3. we don't know if they can be solved polynomially at all

This class is a gray area between the P-class and the E-class. It will be studied in this chapter.

[Back to Top](http://www.mutah.edu.jo/userhomepages/CS252/npcomplete.html#top)

**II. Definition of NP**

1. Definition 1 of NP: A problem is said to be Nondeterministically Polynomial (NP) if we can find a nodeterminsitic Turing machine that can solve the problem in a polynomial number of nondeterministic moves.
2. For those who are not familiar with Turing machines, two alternative definitions of NP will be developed.
3. Definition 2 of NP: A problem is said to be NP if
   1. its solution comes from a finite set of possibilities, and
   2. it takes polynomial time to verify the correctness of a candidate solution
4. Remark: It is much easier and faster to "grade" a solution than to find a solution from scratch.
5. We use NP to designate the class of all nondeterministically polynomial problems.
6. Clearly, P is a subset of NP
7. A very famous open question in Computer Science:
   1. P = NP ?
8. To give the 3rd alternative definition of NP, we introduce an imaginary, non-implementable instruction, which we call "choose()".
9. Behavior of "choose()":
   1. if a problem has a solution of N components, choose(i) magically returns the i-th component of the CORRECT solution in constant time
   2. if a problem has no solution, choose(i) returns mere "garbage", that is, it returns an uncertain value.
10. An NP algorithm is an algorithm that has 2 stages:
    1. The first stage is a guessing stage that uses choose() to find a solution to the problem.
    2. The second stage checks the correctness of the solution produced by the first stage. The time of this stage is polynomial in the input size n.
11. Template for an NP algorithm:

begin

/\* The following for-loop is the guessing stage\*/

for i=1 to N do

X[i] := choose(i);

endfor

/\* Next is the verification stage \*/

Write code that does not use "choose" and that

verifies if X[1:N] is a correct solution to the

problem.

end

* Remark: For the algorithm above to be polynomial, the solution size N must be polynomial in n, and the verification stage must be polynomial in n.
* Definition 3 of NP: A problem is said to be NP if there exists an NP algorithm for it.
* Example of an NP problem: The Hamiltonian Cycle (HC) problem
  1. Input: A graph G
  2. Question: Goes G have a Hamiltonian Cycle?
* Here is an NP algorithm for the HC problem:

begin

/\* The following for-loop is the guessing stage\*/

for i=1 to n do

X[i] := choose(i);

endfor

/\* Next is the verification stage \*/

for i=1 to n do

for j=i+1 to n do

if X[i] = X[j] then

return(no);

endif

endfor

endfor

for i=1 to n-1 do

if (X[i],X[i+1]) is not an edge then

return(no);

endif

endfor

if (X[n],X[1]) is not an edge then

return(no);

endif

return(yes);

end

* The solution size of HC is O(n), and the time of the verification stage is O(n2). Therefore, HC is NP.
* The K-clique problem is NP
  1. Input: A graph G and an integer k
  2. Question: Goes G have a k-clique?
* Here is an NP algorithm for the K-clique problem:

begin

/\* The following for-loop is the guessing stage\*/

for i=1 to k do

X[i] := choose(i);

endfor

/\* Next is the verification stage \*/

for i=1 to k do

for j=i+1 to k do

if (X[i] = X[j] or (X[i],X[j]) is not an edge) then

return(no);

endif

endfor

endfor

return(yes);

end

* The solution size of the k-clique is O(k)=O(n), and the time of the verification stage is O(n2). Therefore, the k-clique problem is NP.

[Back to Top](http://www.mutah.edu.jo/userhomepages/CS252/npcomplete.html#top)

**III. Focus on Yes-No Problems**

* Definition: A yes-no problem consists of an instance (or input I) and a yes-no question Q.
* The yes-no version of the HC problem was described above, and so was the yes-no version of the k-clique problem.
* The following are additional examples of well-known yes-no problems.
* The subset-sum problem:
  + Instance: a real array a[1:n]
  + Question: Can the array be partitioned into two parts that add up to the same value?
* The satisfiability problem (SAT):
  + Instance: A Boolean Expression F
  + Question: Is there an assignment to the variables in F so that F evaluates to 1?
* The Treveling Salesman Problem   
  **The original formulation**:
  + Instance: A weighted graph G
  + Question: Find a minimum-weight Hamiltonian Cycle in G.

**The yes-no formulation**:

* + Instance: A weighted graph G and a real number d
  + Question: Does G have a Hamiltonian cycle of weight <= d?

[Back to Top](http://www.mutah.edu.jo/userhomepages/CS252/npcomplete.html#top)

**IV. Reductions and Transforms**

* Notation: If P stands for a yes-no problem, then
  + IP: denotes an instance of P
  + QP: denotes the question of P
  + Answer(QP,IP): denotes the answer to the question QP given input IP
* Let P and R be two yes-no problems
* Definition: A transform (that transforms a problem P to a problem R) is an algorithm T such that:
  + The algorithm T takes polynomial time
  + The input of T is IP, and the output of T is IR
  + Answer(QP,IP)=Answer(QR,IR)
* Definition: We say that problem problem P reduces to problem R if there exists a transform from P to R.

[Back to Top](http://www.mutah.edu.jo/userhomepages/CS252/npcomplete.html#top)

**V. NP-Completeness**

* Definition: A problem R is NP complete if
  1. R is NP
  2. Every NP problem P reduces to R
* An equivalent but casual definition: A problem R is NP-complete if R is the "most difficult" of all NP problems.
* Theorem: Let P and R be two problems. If P reduces to R and R is polynomial, then P is polynomial.
* Proof:
  1. Let T be the transform that transforms P to R. T is a polynomial time algorithm that transforms IP to IR such that

Answer(QP,IP) = Answer(QR,IR)

* 1. Let AR be the polynomial time algorithm for problem R. Clearly, A takes as input IR, and returns as output Answer(QR,IR)
  2. Design a new algorithm AP as follows:   
     Algorithm AP(input: IP)   
     begin   
             IR := T(IP);  
             x := AR(IR);   
             return x;  
     end
  3. Note that this algorithm AP returns the correct answer Answer(QP,IP) because x = AR(IR) = Answer(QR,IR) = Answer(QP,IP).
  4. Note also that the algorithm AP takes polynomial time because both T and AR   
     Q.E.D.
* The intuition derived from the previous theorem is that if a problem P reduces to problem R, then R is at least as difficult as P.
* Theorem: A problem R is NP-complete if
  1. R is NP, and
  2. There exists an NP-complete problem R0 that reduces to R
* Proof:
  1. Since R is NP, it remain to show that any arbitrary NP problem P reduces to R.
  2. Let P be an arbitrary NP problem.
  3. Since R0 is NP-complete, it follows that P reduces to R0
  4. And since R0 reduces to R, it follows that P reduces to R (by transitivity of transforms).

Q.E.D.

* The previous theorem amounts to a strategy for proving new problems to be NP complete. Specifically, to problem a new problem R to be NP-complete, the following steps are sufficient:
  1. Prove R to be NP
  2. Find an already known NP-complete problem R0, and come up with a transform that reduces R0 to R.
* For this strategy to become effective, we need at least one NP-complete problem. This is provided by Cook's Theorem below.
* Cook's Theorem: SAT is NP-complete.

[Back to Top](http://www.mutah.edu.jo/userhomepages/CS252/npcomplete.html#top)

**VI. NP-Completeness of the k-Clique Problem**

1. The k-clique problem was laready shown to be NP.
2. It remain to prove that an NP-complete problem reduces to k-clique
3. Theorem: SAT reduces to the k-clique problem
4. Proof:
   1. Let F be a Boolean expression.
   2. F can be put into a conjunctive normal form: F=F1F2...Fr   
      where every factor Fi is a sum of literals (a literal is a Bollean variable or its complement)
   3. Let k=r and G=(V,E) defined as follows:   
      V={<xi,Fj> | xi is a variable in Fj}   
      E={(<xi,Fj> , <ys,Ft>) | j !=t and xi != ys'}   
      where ys' is the complement of ys
   4. We prove first that if F is satisfiable, then there is a k-clique.
   5. Assume F is satisfiable
   6. This means that there is an assignment that makes F equal to 1
   7. This implies that F1=1, F2=1, ... , Fr=1
   8. Therefore, in every factor Fi there is (at least) one variable assigned 1. Call that variable zi
   9. As a result, <z1,F1>, <z2,F2>, ... , <zk,Fk> is a k-clique in G because they are k distinct nodes, and each pair (<zi,Fi> , <zj,Fj>) forms an edge since the endpoints come from different factors and zi != zj' due to the fact that they are both assigned 1.
   10. We finally prove that if G has a k-clique, then F is satistiable
   11. Assume G has a k-clique <u1,F1>, <u2,F2>, ... , <uk,Fk> which are pairwise adjacent
   12. These k nodes come the k fifferent factors, one per factor, becuae no two nodes from the same factor can be adjacent
   13. Furthermore, no two ui and uj are complements because the two nodes <ui,Fi> and <uj,Fj> are adjacent, and adjacent nodes have non-complement first-components.
   14. As a result, we can consistently assign each ui a value 1.
   15. This assignment makes each Fi equal to 1 because ui is one of the additive literals in Fi
   16. Consequently, F is equal to 1.

Q.E.D.

* An illustration of the prrof will be carried out in class on   
  F=(x1 + x2)(x1' + x3)(x2 + x3')

[Back to Top](http://www.mutah.edu.jo/userhomepages/CS252/npcomplete.html#top)

1. These k nodes come the k fifferent factors, one per factor, becuae no two nodes from the same factor can be adjacent
2. Furthermore, no two ui and uj are complements because the two nodes <ui,Fi> and <uj,Fj> are adjacent, and adjacent nodes have non-complement first-components.
3. As a result, we can consistently assign each ui a value 1.
4. This assignment makes each Fi equal to 1 because ui is one of the additive literals in Fi
5. Consequently, F is equal to 1.