Complexity Theory and NP-Completeness– The first few steps or: Can a computer solve all problems? Efficiently?

Appie van de Liefvoort©, CSEE, UMKC
Man Kong©, EECS, KU

November 13, 2018

Contents

1 Introduction

It has been said that there are at least two ways to catch a fly: with honey and with vinegar. One of these methods is much more effective in catching a fly, however. Such is the story of most problems in computer science: there are several methods or algorithms for their solution and some algorithms are more efficient than others. There are a number of problems, however, where the choice of algorithm does not seem to matter much: the worst case time complexity is terrible, no matter how you turn it. This is the focus of this section: Can a problems be solved using a computer? And if so, can it be solved efficiently?

So just this question by itself raises more questions: What is the definition of a problem? What is a computer? What do you mean with efficient? What do you mean when you say: “Can this problem be solved efficient with a computer”? And similarly, what does it mean when you say: “There is no efficient solution to this problem.”? Does it mean that you currently do not know of any algorithm that is efficient, but in the future that may be such an algorithm? OR: I can prove that an efficient algorithm simply does not exist. The latter involves lower bound theory: Every search takes at least n comparisons in the worst case, so $T^W(n) = \Omega(n)$ for any algorithm. Similarly, any sorting algorithm is $T^W(n) = \Omega(n \lg n)$

We will have to clarify these terms. Solving problems with a computer means: using an algorithm. So the answer to the question: “Can this problem be solved efficient with a computer’?’ can simply be answered by presenting an algorithm that is both correct and efficient, whatever that means. But if the problem can be solved, can we say: “There is no efficient solution to this problem”? No, we can only say: “There is no efficient solution presently known to this problem”? So that brings us to algorithms that may not yet exist. Algorithms that will be discovered/constructed/created next year, or in 10 years, or in 100 years. When you want to discuss algorithms that are yet to be derived (or discovered, or invented, or ...), then there is not much we can say at the algorithmic level. Instead, we need to focus on the problem that any new algorithm purports to solve. So we need to focus on the problems and problem statements. What is a problem? Can give a proper definition of a problem? We have solved several CS-related problems before: Finding problems, Construction Problems, Evaluating Problems, Optimization Problems, and so on. If we introduce a problem as a formal abstract construct, (and we will introduce a decision problem formally below) then we can analyze a problem, propose algorithms that solve them, and say something about the problem and the algorithms that have been introduced to solve it. Let us slowly build on the concept.

REMARK 1:
A *problem* or *problem formulation* is a generic problem statement using unspecified parameters to denote appropriate constants. An *instance of the problem* (instance, in short) refers to the particular formulation with all the parameters (constants) substituted with their actual values. For example: “Sort an array of n integers” is a generic problem, whereas “Sort this array of 277 integers” is an instance of the generic sorting problem.

REMARK 2:
Problems may have one unique solution (e.g. a sorted array), may have several correct solutions (e.g. a topological sort in a directed graph), or may not have a solution at all. Thus we have a *solution space* of (feasible, optimal) *solutions*. *Size of the problem instance*, n, should be understood intuitively; observe, however, that in general the size may depend not only on the number of variables and constraints, but also on the numerical values of the bounds used in the formulation (as e.g. the knapsack size in the knapsack problem)

REMARK 3:
Informally, the problems we consider fall into three categories: Those for which we have not found an algorithm, those for which there are some algorithms that always solve the problem in reasonable time, and those problem that can be solved and for which there is no such guarantee. For instance, consider an array of $2n$ integers. A problem that can be solved in reasonable time is: ”Is this particular input such that the sum of the first n integers is equal the sum of the last n integers?”, whereas the question ”Is there a particular permutation such that the sum of the first n integers is equal to the sum of the last n integers?” is much harder to answer. Notice that verifying that the sum of the first n integers is equal to the sum of the last n integers can be done in linear time, which is of course polynomial.

REMARK 4:
Often the problem can be reformulated as: ”Does a particular instance of the problem have property X?” or ”Is there an element in the solution space of a particular instance that has property X?” It is assumed that checking whether or not an instance has the property can be done by an algorithm with a worst case asymptotic time complexity that is bounded above by a polynomial. (The term ”time complexity” in this section will refer to the worst case.) Thus the first category of problems has a polynomial time complexity, whereas the second needs to search for an solution in the solution space of the considered instance, and has a time complexity depending in general on the size of this solution space. If this size is exponential with the problem (instance) size, then any algorithm based on brute-force checking of all possibilities will have a time complexity that is at least exponential. At the same time it appears that such an exhaustive search can be avoided by using a smart algorithm.

The purpose of this chapter is not to give a thorough and complete introduction to complexity theory, but rather to develop an appreciation and intuition for the direction taken. For a more detailed and rigorous presentation, see e.g. references [7, 9, 11].

2 Complexity of a Problem

Complexity theory addresses problems, the known algorithms that solve them, and the potential to find/discover/develop new algorithms that outperform the known algorithms. The theory must be very carefully built since it must be valid for all problems and algorithms, whether or not these problems or algorithms have already been identified, or are yet to be discovered. Although the theory has to apply to all problems, it is especially applicable to problems for which there are, as yet, no efficient solutions known. So let us extend the notion of complexity of an algorithm to the complexity of a problem.

The notion of the *complexity of a problem* is defined similar to the notion of the complexity of an algorithm that solves the given problem. Except that there are perhaps multiple algorithms that solve the problem, and that there are algorithms out there that are yet to be discovered. So given a problem with input of size n.

Let us take as example the definition of a linear problem: (Other complexities are defined similarly.)
1. There is an algorithm X that solves the problem,
2. It’s worst case time complexity is $T_X^W(n) = \mathcal{O}(n)$ *and*
3. if any other algorithm Y that also solves the problem (already discovered or yet to be discovered) has complexity $T_Y^W(n) = \Omega(n)$.

Algorithmic gap

Criterion 3. is not consistently included in the definition. We will, and call a problem “at most linear” if only cases 1. and 2. apply. Showing a lower bound is considered “Lower Bound Theory” and Lower Bounds are generally only trivially known. Problems for which the best lower bound is definitely smaller than the best known published algorithm, that is $T_X^W(n) = \mathcal{O}(T_{\text{lower bound}}(n))$, are said to have an algorithmic gap. This gets to the core of computer science:

If a problem X is in the algorithmic gap then Close the gap by finding a “better” algorithm, or prove that such a better algorithms does not exist.

Proving that a better algorithms does not exist, means it does not exist now, and not over 10 or 50 years. This means: find a higher lower bound, and make the gap smaller. There are many problems that have an algorithmic gap, and there are many algorithms that do not have an algorithm gap. Examples: Selection from n elements is a linear problem, comparison based sorting n elements is an $n \log n$ problem, the (standard) towers of Hanoi with n rings is exponential, and so on. The disadvantage of such a classification is that there are many problems that are not classified yet. Take for instance the 0-1 knapsack problem or the traveling salesman problem (find a tour of smallest total weight in a graph with integer edge weights that visits every node exactly once). There are algorithms that solve these problems in exponential amount of time, but the lower bound theory only demands ”at least polynomial”. Such problems are said to have an algorithmic gap and complexity theory attempts to close this gap; either by developing faster algorithms or by providing a higher lower bound.

3 Decision Problems

The theory is presented for *decision* problems. These are problems for which the solution is a **yes/no** answer, such as ”Is this array sorted?”, ”Is element a at location $A[123]$?” The collection of decision problems will be denoted as $\mathcal{D}$, and a single decision problem will be denoted as a $\mathcal{D}$-problem (or just as a problem if it is clear from the context). Define:

$$\mathcal{D} = \{\text{the set of decision problems, they have } \textbf{yes/no} \text{ answers. } \} \tag{1}$$

Note that all optimization problems have a version that is a decision problem, such as ”Is there a path from A to B whose total cost is less than K?” An optimization problem can often be solved by solving a sequence of decision problems, such as in

begin $k := 1$; **while** *solveDecisionProblemX*(k) **do** $k := k + 1$; **end**.

3.1 Gallery of Decision Problems

Many problems are themselves decision problems, and others can be given in an Decision-like form. Below we give some examples. First, we start with some easy ones, that have algorithms that we can imagine having seen before. These are followed by some classical examples that will be used to study complexity theory.

1: Find item in a data structure

Situation Find/locate a given element x in a given data structure (be it array, list, queue, tree, graph, and so on.)

Decision Formulation Is the element x in the data structure? (be it array, list, queue, tree, graph, and so on.)

2: Sort an Array

Situation Sort a given array.

Decision Formulation Is the array sorted?

3: Construct Special data structure

Situation Construct an Optimal Binary Search Tree

Decision Formulation Is the given search tree an optimal binary search tree?

4: Construct Special data structure - again

Situation Construct an DFS-spanning tree

Decision Formulation Is the given tree the an DFS-spanning tree?

5: Construct Special data structure - again

Situation Construct an topological sort

Decision Formulation Is the given sequence a topological sort?

6: 0-1 Knapsack Problem

Situation Given a knapsack that can hold items with a maximal combined weight W and n items, each with a weight w_i and a value v_i, all integers.

Optimization Formulation What is the maximal combined value of items such that their combined weight is at most W?

Decision Formulation Is there a subset of items with a combined weight limited by W such that the combined value is V or larger? (The value V is an additional input).

7: Changing Coin Problem

Situation Given a set of an unlimited number of coins with a finite set of denominations d_i and a desired value V

Optimization Formulation What is the smallest number of coins needed such that the combined value of these coins equals V.

Decision Formulation Can k coins be taken with a combined value equals V. (The value of k is an additional input).

8: Maximum Independent Set

Situation Given a graph $G = (V, E)$ with n nodes and m edges.

Optimization Formulation Find the largest subset of V that forms an Independent Set.

Decision Formulation Is there a subset of V of size k that forms an Independent Set? (The value of k is an additional input).

9: Minimum Node Cover

Situation Given a graph $G = (V, E)$ with n nodes and m edges.

Optimization Formulation Find the smallest subset of V that forms a Node Cover (i.e. every edge has at least one end-point node in the Node Cover).

Decision Formulation Is there a subset of V of size k that forms a Node Cover? (The value of k is an additional input).

10: Minimum Edge Cover

Situation Given a graph $G = (V, E)$ with n nodes and m edges.

Optimization Formulation Find the smallest subset of E that forms an Edge Cover (i.e. every node is an end-point in at least one edge in the Edge Cover).

Decision Formulation Is there a subset of E of size k that forms an Edge Cover? (The value of k is an additional input).

11: Minimum Set Cover

Situation Given a (finite) universe U of elements, $U = \{e_1, e_2, \ldots e_n\}$ and additionally a set of subsets $S_1, S_2, S_3, \ldots S_m$.

Optimization Formulation Find the smallest number of subsets such that their union equals U. That is, every element of U is in at least one of the chosen subsets.

Decision Formulation Are there k subsets amongst the $S_1, S_2, S_3, \ldots S_m$ such that their union covers U? (The value of k is an additional input).

12: Traveling salesperson

Situation Given a graph $G = (V, E)$ with n nodes and m weighted edges.

Optimization Formulation Is there cycle that visits all the nodes $v_1, \ldots v_n$ exactly once, and such that the total cycle weight is smallest amongst all such cycles?

Decision Formulation Is there cycle that visits all the nodes $v_1, \ldots v_n$ exactly once, and such that the total cycle weight is $\leq k$? (The value of k is an additional input).

13: Subset sum Be careful: there are several flavors of this question in the literature

Situation Given a set S of n integer elements, $S = \{e_1, e_2, \ldots e_n\}$.

Decision Formulation Can the set be partitioned into 2 subsets A and B, such that $\sum_{a \in A} a == \sum_{b \in B} b$?

14: SubArray sum

Situation Given an array set C with n integer elements, $C = [c_1, c_2, \ldots c_n]$.

Decision Formulation Can the array be split into two parts, $A = [c_1, c_2, \ldots c_{k-1}]$ and $B = [c_k, \ldots c_n]$, such that $\sum_{i<k} c_i == \sum_{j \geq k} c_j$?

15: Linear programming

Situation Given a list of m linear inequalities with real-valued coefficients over n variables $u_1, ..., u_n$ (for example, the j^{th}

linear inequality has the form $a_{j,1}u_1 + a_{j,2}u_2 + ... + a_{j,n}u_n \leq c_j$ for some coefficients $a_{j,1}, ..., a_{j,n}$ and c_j).
Decision Formulation Is there an assignment of real numbers to the variables $u_1, ..., u_n$ that satisfies all the inequalities.

16: Integer programming
Situation Given a list of m linear inequalities with integer-valued coefficients over n variables $u_1, ..., u_n$ (for example, the j^{th} linear inequality has the form $a_{j,1}u_1 + a_{j,2}u_2 + ... + a_{j,n}u_n \leq c_j$ for some integer coefficients $a_{j,1}, ..., a_{j,n}$ and c_j).
Decision Formulation Is there an assignment of integers to the variables $u_1, ..., u_n$ that satisfies all the inequalities.

17: Satisfiability
Situation Given a Boolean expression $f(\cdot)$ with n literals (and m clauses).
Decision Formulation Is there an assignment of TRUE/FALSE to these literals that is consistent and such that the Boolean expression $f(\cdot) = TRUE$.

18: Graph isomorphism
Situation Given two graph with two $n \times n$ adjacency matrices **A** and **B**.
Decision Formulation Do these two adjacency matrices define the same graph, up to renaming the vertices?

4 Deterministic and Non-Deterministic Turing Machines

As already mentioned, complexity theory addresses problems and algorithms that solve them, both known algorithms as well as algorithms that have yet to be discovered. The only common framework for these algorithms are the machines that run them. Again, machines that have been build already and are operational, as well as machines that have yet to be designed and build. The theory must therefore be based on the same theoretical foundation upon which all current machines are build: the deterministic machine, where each next step is completely determined by the current internal state of the machine and the content of the cell on the tape at the location of the R/W head. This deterministic machine is reviewed below and extended to include non-deterministic behavior.

The following definitions of a deterministic and a non-deterministic (Turing) machine are somewhat loose, but suffice for the current presentation. A *deterministic (Turing) machine*,

1 can read and write on a (double sided) infinite tape,
2 can scan one cell on the tape at any one time,
3 can make a finite number of operations on the tape (e.g. write 0/1),
4 can move either to the left or to the right on the tape,
5 has inside a program which tells the machine precisely (i.e. deterministically), where to go and what to do next. In particular, there is only one thing to do, and it is well defined.

Every currently existing computer is a realization of such a deterministic machine. These are extended to a *non-deterministic (Turing) machine*, which has the identical capabilities as 1, 2, 3, and 4 above. Furthermore, capability 5 is replaced and three others are added. Thus, in addition to capabilities 1, 2, 3, and 4, a non-deterministic machine has three additional statements:

5' `CHOICE`: has inside a program which *sometimes* tells the machine precisely where to go and what to do next (the deterministic part). At *other times* there are a finite number (say n) places to go to and/or things to do next. At these times, the machine instantly, and with no computational cost, clones itself into n machines, and continues each cloned version with a different choice. We will write `CHOICE`(S) for these steps, with S indicating the possible choices.
6 There is no communication between these cloned versions,
7 `FAILURE`: If a cloned version realizes that it can not solve the problem, it executes the statement `FAILURE`, which causes it to cease operations, without interrupting the other cloned versions. The `FAILURE` statement carries an $\mathcal{O}(1)$ (i.e. bounded by a constant) computational cost.

8 `SUCCESS`: If a cloned version realizes that it can solve the problem successfully, it executes the statement `SUCCESS`, which causes a **yes** to be printed and causes all other cloned versions to cease operations (with failure). The `SUCCESS` statement also carries an $\mathcal{O}(1)$ computational cost.

An algorithm for a (non-)deterministic machine is referred to as a (non-)deterministic algorithm. A number of remarks are appropriate.

REMARK 5:
All existing computers are realizations of the deterministic machine, and non-deterministic machines are not (yet?) realized, even after many years of trying and billions of research dollars.

REMARK 6:
There are several other formal frameworks that have been used, such as Finite State Machines, Automatons, Language recognizers, and others. Their differences for this initial introduction are not critical. We are setting the general tone of the conversation, and build on intuition.

REMARK 7:
A non-deterministic algorithm *affirms* a problem if the algorithm executes the `SUCCESS` statement, in other words that the **yes** has been printed (and a solution has been generated). An algorithm can not solve the problem if none of the clones can execute a `SUCCESS` statement, and a **yes** can not be generated. Thus our non-deterministic algorithm cannot be used to decide between **yes** and **no**, it can only generate **yes** when the problem has been solved. This appears to be a playing with words, but the difference is very real: the algorithm can report only positive result by executing the `SUCCESS` statement. Thus as soon as there is a single clone that finds it can solve the problem, all other clones cease their operation. Negative results cannot be reported, because it would imply that clones can communicate somehow, or that there is a ”master” that keeps track of the number of clones that have died with failure. (The cloning is similar the **fork** construct in a UNIX-like language, but in our developments, there is no **join** construct.)

In particular: The NTM is not symmetric for yes and no. This is the reason that we no not use the word “solving” (since this implies symmetry) and distinguish between *affirming* and *deciding*. The latter will be used for a problem that can be affirmed AND that its complementary problem also can be affirmed.

REMARK 8:
Please refer to capability 7 of a non-deterministic machine: ”If a cloned version realizes that it can not solve the problem, it executes the statement `FAILURE`, which causes it to cease operations, without interrupting the other cloned versions.” Negative results can thus not be reported in this set-up, because there is no ”master” that can keep track of the number of clones that ”failed”. If negative results need to be acknowledged and reported, then the complement of the problem needs to be defined. This is the start of ”co-$\mathcal{NP}$” theory, which is not addressed in this appendix.

REMARK 9:
There is thus a difference between “solving” and “affirming.” Previous versions of these notes have used “solving” only, and I am now changing some language to reflect more correct “affirm” but some inconsistencies may still be present. In future, there will be two classes of problems: Affirmable problems (only YES) and Decidable problems, (both YES and NO answers).

REMARK 10:
We have taken this particular approach to non-deterministic machines. Others use a slightly different approach, and use words like: one-way solvable, semi-solvable, accept, enumerable, recognizable, or some other similar word.

4.1 Examples of non-deterministic decision algorithms

Algorithm Number 1:

```
LINE 100: ND-Algorithm CharAtIndex(array $myArray$, int $n$, char $MyChar$, index $MyIndex$)
LINE 110: if $MyArray[MyIndex] == MyChar$
LINE 120:     then SUCCESS
LINE 130:     else FAILURE
LINE 140: end-if
```

Algorithm Number 2:

```
LINE 100: ND-Algorithm CharInArray(array $myArray$, int $n$, char $MyChar$)
LINE 110: if $myArray$[CHOICE($1 \ldots n$)] == $MyChar$
LINE 120:     then SUCCESS
LINE 130:     else FAILURE
LINE 140: end-if
```

Algorithm Number 3:

```
LINE 100: ND-Algorithm TwoInArray(array $myArray$, int $n$)
LINE 110: $i \leftarrow$ CHOICE($1 \ldots n$)
LINE 120: $j \leftarrow$ CHOICE($1 \ldots n$)
LINE 130: if $i \neq j$ and $myArray[i] == myArray[j]$)
LINE 140:     then SUCCESS
LINE 150:     else FAILURE
LINE 160: end-if
```

Algorithm Number 4:

```
LINE 100: Algorithm-ND NotSortedArray(array $myArray$, int $n$)
LINE 110: $i \leftarrow$ CHOICE($1 \ldots n-1$)
LINE 120: if $myArray[i] > myArray[i+1]$)
LINE 130:     then SUCCESS
LINE 140:     else FAILURE
LINE 150: end-if
```

Algorithm Number 5:

```
LINE 100: ND-Algorithm SortedArray(array $myArray$, int $n$)
LINE 110: for $i$ from $1$ to $n-1$ do
LINE 120: if $myArray[i] > myArray[i+1]$)
LINE 130:     then FAILURE
LINE 140: end-if
LINE 150: SUCCESS
```

Algorithm Number 6:

```
LINE 100: ND-Algorithm $k$-Colorability(graph $myGraph$, int $m, n$)
LINE 110: for $i$ from $1$ to $n$ do
LINE 120:     $ColorLabel[i] \leftarrow$ CHOICE$(1 \ldots k)$
LINE 130: end-for
LINE 140: for all $(u, v) \in EdgeSet$ do
LINE 150:     if $ColorLabel[u] == ColorLabel[v]$ then FAILURE end-if
LINE 160: end for
LINE 170: SUCCESS
```

Algorithm Number 7:

```
LINE 100: ND-Algorithm NDSortingArray(array $myArray$, int $n$)
LINE 110: SWAP($MyArray[1]$, $MyArray[$ CHOICE$(1 \ldots n)]$ )
LINE 120: for $i$ from $2$ to $n - 1$ do
LINE 130:     SWAP($MyArray[i]$, $MyArray[$ CHOICE$(i \ldots n)]$ )
LINE 140:     if $myArray[i - 1] > myArray[i]$) then FAILURE end-if
LINE 150: end-do
LINE 160: if SortedArrayHasProperty($myArray$) then SUCCESS else FAILURE end if
```

Algorithm Number 8:

```
LINE 100: ND-Algorithm NDPermutationsArray(array $myArray$, int $n$)
LINE 110: for $i$ from $1$ to $n$ do
LINE 120:     SWAP($MyArray[i]$, $MyArray[$ CHOICE$(i \ldots n)]$ )
LINE 130: end-do
LINE 140: if PermutationHasProperty($myArray$) then SUCCESS else FAILURE end if
```

Algorithm Number 9:

```
LINE 100: ND-Algorithm NDCombinationsPermuteArray(array $myArray$, int $n$, int $k$)
LINE 110: for $i$ from $1$ to $k$ do
LINE 120:     SWAP($MyArray[i]$, $MyArray[$ CHOICE$(i \ldots n)]$ )
LINE 130: end-do
LINE 140: if CombinationHasProperty($myArray, k$) then SUCCESS else FAILURE end if
```

Algorithm Number 10:

```
LINE 100: ND-Algorithm ND-SubSets(set $myS$, int $n$)
LINE 110: $MySub \leftarrow \emptyset$
LINE 120: for al $s \in myS$ do
LINE 130:     if CHOICE(TRUE, FALSE)
LINE 140:         then $MySub \leftarrow MySub \cup \{ s \}$
LINE 150:     end-if
LINE 160: end-for-all
LINE 170: if SubSetHasProperty($MySub$) then SUCCESS else FAILURE end if
```

Algorithm Number 11:

```
LINE 100: ND-Algorithm ND-$k$-PARTITION(set $myS$, int $n$)
LINE 110: for $i$ from $1$ to $k$ do $MyPart[i] \leftarrow \emptyset$ end-for
LINE 120: for al $s \in myS$ do
LINE 130:     $b \leftarrow$ CHOICE$(1 \dots k)$
LINE 140:     $MyPart[b] \leftarrow MyPart[b] \cup \{ s \}$
LINE 150: end-for-all
LINE 160: if PartitionHasProperty($MyPart$) then SUCCESS else FAILURE end if
```

5 Suggested Exercises

For all the examples above, and for all the exercises below: Find the highest number of clones that were active at the same time, and find the time complexity for the algorithms. For the algorithms below: What are the following ND-Algorithms doing? Are they correct? If they are correct, are they generating a discrete or combinatorial structure? What would it be?

SE.1: Algorithm Number 12:

```
LINE 100: ND-Algorithm ND-12(graph $myGraph = (V, E)$)
LINE 110: $X \leftarrow \emptyset$    $n \leftarrow V$.size    $m \leftarrow E$.size
LINE 120: for all $v_i \in V$ do if CHOICE( {True, False} ) then $X \leftarrow X \cup \{v_i\}$ end-if  end-for
```

SE.2: Algorithm Number 13:

```
LINE 100: ND-Algorithm ND-13(graph $myGraph = (V, E)$ )
LINE 110: $X \leftarrow \emptyset$    $n \leftarrow V$.size    $m \leftarrow E$.size
LINE 120: for all $v_i \in V$ do $X \leftarrow X \cup$ CHOICE$(\{\{v_i\}, \emptyset\})$ end-for
```

SE.3: Algorithm Number 14:

```
LINE 100: ND-Algorithm ND-14(graph $myGraph = (V, E)$, int $k$)
LINE 110: $X \leftarrow \emptyset$    $n \leftarrow V$.size    $m \leftarrow E$.size
LINE 120: for $j$ from $1$ to $k$ do $X \leftarrow X \cup$ CHOICE$(V)$ end-for
```

SE.4: Algorithm Number 15:

```
LINE 100: ND-Algorithm ND-15(graph $myGraph = (V, E)$, int $k$)
LINE 110: $X \leftarrow \emptyset$    $n \leftarrow V$.size    $m \leftarrow E$.size
LINE 120: for $j$ from $1$ to $k$ do $X \leftarrow X \cup$ CHOICE$(V \cap X^{c})$ end-for
```

SE.5: Algorithm Number 16:

```
LINE 100: ND-Algorithm ND-16(graph $myGraph = (V, E)$, int $k$)
LINE 110: $X \leftarrow \emptyset$    $n \leftarrow V$.size    $m \leftarrow E$.size
LINE 120: for all $v_i \in V$ do $Label[v_i] \leftarrow$ CHOICE$(\{1, 2, \dots k\})$ end-for
```

SE.6: Algorithm Number 17:

```
LINE 100: ND-Algorithm ND-17(graph $myGraph = (V, E)$, int $k$)
LINE 110: $X \leftarrow \emptyset$        $n \leftarrow V$.size        $m \leftarrow E$.size
LINE 120: for all $(u, v) \in E$ do $Label[(u, v)] \leftarrow$ CHOICE($\{1, 2, \ldots k\}$) end-for
```

SE.7: Algorithm Number 18:

```
LINE 100: ND-Algorithm ND-18(graph $myGraph = (V, E)$, node $Start$)
LINE 110: // Precondition: $ColorLabel[Start] == $ WHITE
LINE 120: $n \leftarrow V$.size        $m \leftarrow E$.size
LINE 130: $ColorLabel[Start] \leftarrow$ GRAY
LINE 140: $w \leftarrow$CHOICE($\{$ nodes adjacent to $Start\}$ )
LINE 150: if ($ColorLabel[w] \neq$ WHITE) then FAILURE end if
LINE 160: Recursive Call ND-18 ($myGraph = (V, E), w$)
```

SE.8: Now write your own **ND-Algorithm**s for all the decision problems presented in the section "Gallery of Decision Problems", except the construction problems that are surprisingly hard (is a search tree an OBST, is it a DFS tree, is it a topological sort and such others).

SE.9: (graduate students) Now write your own **ND-Algorithm**s for all the decision problems you skipped, be sure to include one or two construction decision problems.

6 Problems with an Affirmative Algorithm

There is a large number of decision problems for which algorithms have been designed that affirms and/or solves them. But there is no difference between deterministic machines and non-deterministic machines as far as the set of problems they each can solve. Define:

$$\mathcal{C}_{\mathcal{D}} = \{X \in \mathcal{D} : \text{there is a deterministic algorithm A that affirms } X.\} \tag{2}$$

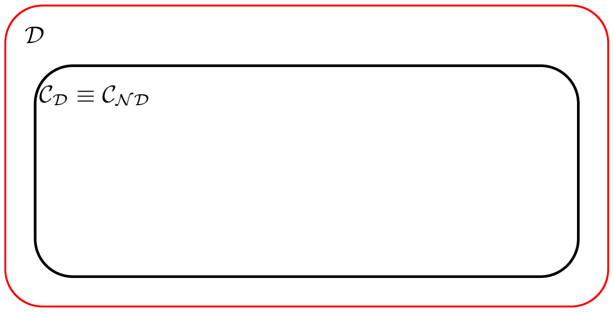
and similarly:

$$\mathcal{C}_{\mathcal{ND}} = \{X \in \mathcal{D} : \text{there is a non-deterministic algorithm A that affirms } X.\} \tag{3}$$

Notice that a non-deterministic machine with only deterministic steps is in essence a deterministic machine, so that $\mathcal{C}_{\mathcal{D}} \subseteq \mathcal{C}_{\mathcal{ND}}$. The reverse is also true: $\mathcal{C}_{\mathcal{ND}} \subseteq \mathcal{C}_{\mathcal{D}}$. Consider the execution path of a non-deterministic machine. At each deterministic step, there is precisely one thing to do next, so there is only one operation following the current one on the execution path. At each non-deterministic step, there is more than one, but a finitely bounded number of possible operations on the execution path. So the execution path of a non-deterministic algorithm for all clones concurrently can be envisioned as a general tree, where each node has a finite number of children. A deterministic algorithm can now be written that mimics a level-order traversal (=breath first search) of this general execution tree. Thus:

$$\boxed{\boxed{\mathcal{C}_{\mathcal{D}} \equiv \mathcal{C}_{\mathcal{ND}}}} \tag{4}$$

This means that every problem that can be solved on a non-deterministic machine can also be solved on a deterministic machine. There is thus no difference between these two machines when considering the kinds of problems it can solve. Rather the difference is in the time it might need to execute them.


$\mathcal{D}$
$\mathcal{C}_{\mathcal{D}} \equiv \mathcal{C}_{\mathcal{ND}}$

REMARK 11:
So we now have a collection of problems for which there *is* a solution. But what does "*is*" mean in this context? It can be one of three things:

1. "*is*, and we can read all about it." We will refer to these algorithms as being published.
2. "*is*, and will be discovered and described in the future."
3. "*is*, and will never be."

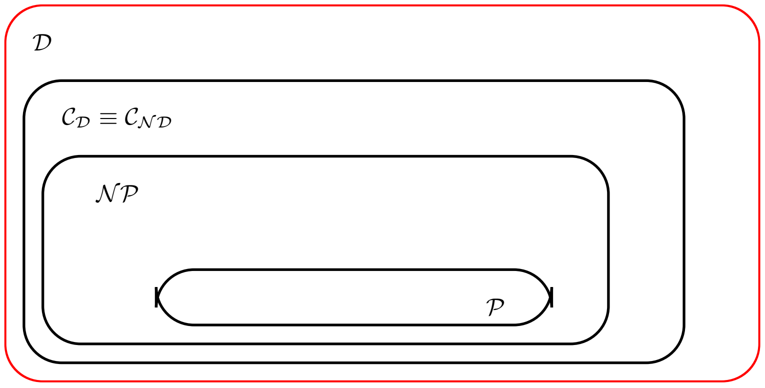
We will try to avoid any ambiguous statements in this regard, and use "published algorithm" or "known algorithm". Even though in fact it may not have been published. Particularly, several of such algorithms are sometimes simply assigned as homework and not published.

7 The Classes of Problems Known as $\mathcal{P}$ and $\mathcal{NP}$

So we found that $\mathcal{C}_{\mathcal{D}} \equiv \mathcal{C}_{\mathcal{ND}}$: if a problem can be affirmed on a non-deterministic machine, then it can also be affirmed on a deterministic machine. So perhaps the time complexity is different. Indeed, it is. But before we present the results, we need another big grain of salt: What does time complexity mean for either machine? What does a size of n mean for our machines? If you need to input one variable, then the number of bits involved is the logarithm of its value. So for each variable, there is a logarithmic multiplier involved. In earlier chapters, we have conveniently ignored these, and we still would like to ignore them. To make that happen, we could/should be much more careful with our asymptotic notation. Or we could increase the size of our grain of salt: We introduce the class of POL: we say that $T(n) = \mathcal{O}(\text{POL}(n))$ if $T(n) = \mathcal{O}(n^k)$ for some $k > 0$. W also say, rather loosely, that an algorithm is POLynomial if indeed $T(n) = \mathcal{O}(\text{POL}(n))$ for this algorithm. This means that we can multiply these with logarithmic powers without violating the asymptotic statement. With this new relaxed statement, we can introduce another set of problems that can be affirmed and that can be affirmed in polynomial time:

$$\mathcal{P} = \{X \in \mathcal{C}_{\mathcal{D}} : \text{A deterministic algorithm } A \text{ has been published that affirms } X \text{ and is POLynomial, i.e. } T_A^W(n) = \mathcal{O}(\text{POL}(n)).\} \quad (5)$$

$$\mathcal{NP} = \{X \in \mathcal{C}_{\mathcal{ND}} : \text{A non-det. algorithm } A \text{ has been published that affirms } X \text{ and is POLynomial, i.e. } T_A^W(n) = \mathcal{O}(\text{POL}(n)).\} \quad (6)$$


$\mathcal{D}$
$\mathcal{C}_{\mathcal{D}} \equiv \mathcal{C}_{\mathcal{ND}}$
$\mathcal{NP}$
$\mathcal{P}$

Thus, if a problem X is in the set $\mathcal{P}$, $X \in \mathcal{P}$, then there is a deterministic algorithm that solves X and the length of its execution path is (bounded by a) polynomial. Similarly, if a problem X is of class $\mathcal{NP}$, $X \in \mathcal{NP}$, then there is a non-deterministic algorithm that solves X and its execution tree has an accepting statement whose depth is (bounded by a) polynomial. Again, it is easy to see that $\mathcal{P} \subseteq \mathcal{NP}$. The reverse inclusion, $\mathcal{NP} \subseteq \mathcal{P}$, is however as yet an open question, the most famous open question in computer science. Thus:

$$\boxed{\boxed{\mathcal{P} \subseteq \mathcal{NP}}} \quad \text{but the open question is:} \quad \boxed{\boxed{?\ \ \mathcal{NP} \subseteq \mathcal{P}\ \ ?}} \tag{7}$$

REMARK 12:
If a problem X is in $\mathcal{P}$, that means it can be solved on a deterministic machine in polynomial time. That also means, that it can be done on a non-deterministic machine in polynomial time. This does not imply that it can be done faster on a non-deterministic machine, nor does it exclude this possibility. Consider e.g. the problems "Is the sum of all n elements in a set equal to K?" and "Is there an element x in this set?".

REMARK 13:
Similarly, if problem X is in $\mathcal{NP}$, that means it can be affirmed (solved) on a non-deterministic machine in polynomial time. That also means, that it can be done on a deterministic machine in exponential time by using a level-order traversal of the execution tree. This does not imply that a deterministic algorithm must take exponential time, because it just might be that there is also a deterministic algorithm that solves X in polynomial time. The same two problems can be used as examples here.

REMARK 14:
At this time, many new terms spring up for problems that are in $\mathcal{P}$, like tractable, feasible and such. These terms indicate such problems can be done within reasonable time for a reasonable input size. Data mining often deals with n in the millions, and a quadratic algorithm is very much a challenge.

REMARK 15:
It would appear that a problem being in $\mathcal{P}$ or $\mathcal{NP}$ would indicate efficient or not efficient. By en large, this is true. But consider an algorithm that runs in $T(n) = 2^{\sqrt{n}}$ or even $T(n) = 2^{\sqrt[7]{n}}$. Although these are both $\Omega(\text{POL}(n))$, they are very efficient for rather large values of n.

7.1 Examples of Problems Known to be in $\mathcal{P}$

These are just about all problems for which you have implemented an algorithm: The decision-version of searching, sorting, min, max, minmax, minmin, peaks and valleys, sub-array sum, shortest paths, minimum spanning tree, shortest paths, Optimal Binary Search Tree, Huffman codes, matrix multiplication, matrix chain multiplication, Eulerian cycle, determinants of a matrix, edge cover, fractional Knapsack, Non-Integer Linear Programming, and so on, and so on.

7.2 Examples of Problems NOT Known to be in $\mathcal{P}$

CNF-Sat, 3Sat, 3-Colorability, 0-1 Knapsack, Tower of Hanoi, Integer Linear Programming, Hamiltonian Cycle, Traveling Salesman variations, permanents of a matrix, maximal spanning trees, longest distance, node cover, minimal key-closure, and so on. These will be shown to be in $\mathcal{NPC}$, $\mathcal{NP}$-complete, which we will introduce below,

REMARK 16:
There are still problems that are outside $\mathcal{NP}$: The Towers of Hanoi is an example of a problem that is inherently exponential. The Halting problem is an example of a problem that is not even in $\mathcal{C}_{\mathcal{ND}}$

8 Reducibility Relation Between Problems

Suppose you want to tackle the open question $\mathcal{P} = \mathcal{NP}$? If you believe it is true, you could take a particular problem $X \in \mathcal{NP}$ and find a polynomially bounded deterministic algorithm that solves it. But you must do this for *all* problems in $\mathcal{NP}$, both for problems that are currently already identified as well as problems that have yet to be discovered. So where would you start so that you can be successful? You could try to identify a ”difficult” problem that is still in $\mathcal{NP}$, in fact, you may want one that is ”most difficult”. These ”most difficult” problems together (but still within $\mathcal{NP}$) are known as $\mathcal{NP}$*-complete* problems ($\mathcal{NPC}$, see below for more formal introduction), and each $\mathcal{NPC}$ problem represents in some sense ”all $\mathcal{NP}$ problems together”. You (and all other researchers) could then focus all your energies on any one of the $\mathcal{NPC}$ problems. Let Y be such an $\mathcal{NPC}$ problem. Now, if a deterministic algorithm can be found that solves Y in polynomial time, then $\mathcal{P} = \mathcal{NP}$. On the other hand, if it can be shown that the lower bound for Y is more than polynomial (for instance, it is exponential), then $\mathcal{P} \neq \mathcal{NP}$. Introduce now the notion of (polynomial) *reducibility* which will define a relation between problems X and Y. In a way, this is similar to the relation $x \leq y$ between numbers x and y and the relation $T_X(n) = \mathcal{O}(T_Y(n))$ between timing functions for X and Y.

Let X and Y both be in $\mathcal{C}_{\mathcal{ND}}$. Problem X *(polynomially) reduces* to problem Y, notated as $X \propto Y$, if there is a deterministic algorithm A, whose worst case time complexity is bounded by a polynomial, that transforms any instance of X into a particular instance of Y, such that the result of any algorithm that affirms for the so-created instance of Y is consistent with the result of any algorithm that affirms X when operating on the original instance. Another way to describe this is as follows. Suppose there are non-deterministic algorithms B_p and B_q that affirm problems X and Y, respectively. Suppose that there is a deterministic, polynomial time algorithm A_T that intercepts the input for algorithm B_p and transforms this input into an input for algorithm B_q; it creates an problem instance for problem Y. When this transformed input is given as input to algorithm B_q, then the output of B_q is indistinguishable from the output of algorithm B_p working on the original input I_X, symbolically: $B_Y(A_T(I_X)) \equiv B_X(I_X)$, and in words: B_Y operating on the transformed input $A_T(I_X)$ results in **YES** if and only if B_p operating on I_X results in **YES**.

REMARK 17:
We have not seen examples yet for decision problems, but we have seen examples for traditional problems. For instance, when solving the “baseball pennant race” problem, where we tried to determine whether or not a particular team could still be the sole winner of the league, we constructed an instance of a maximal flow problem, and interpreted the result. Similarly, when designing most reliable networks, we constructed an instance for a shortest path by using the (negative) of the log of the probabilities as weighted distance. After you think about it for a minute: we should all be able to come up with other examples. In other words: reducibility is not a new concept, but we need to give it a new because this concept is important in the theory we are building.

REMARK 18:
The particular form of reducibility we use is known as “Turing-reducibility”, but we do not introduce other forms, and do not refer to Turing.

8.1 The algebra of the Reducibility Inequality

At first glance, this notion of ”reducibility” appears rather awkward, and so it is. It defines a binary relationship between problems X and Y, without resorting to any algorithms that might solve/affirm these problems. The relationship $X \propto Y$ indicates intuitively something like: ”problem X is a special case of problem Y”, or ”problem Y is more general than problem X”, as long as it is understood that there is a polynomial time algorithm involved in showing such specialization/generalization. This is a very common way to solve every day problems. Suppose you need to find the median of a set of natural numbers, then you could first sort these numbers and find the middle. You have transformed (or now called: reduced) a selection problem into a sorting problem. It is an interesting exercise to realize that any problem in $\mathcal{P}$ reduces to any other problem in $\mathcal{P}$. As such, it is true that the problems ”Is there a minimum spanning tree with total cost $\leq M$?” and ”Is 5 less than 8?” are equivalent problems. A beginning programmer (and many experienced ones) might not want to agree.

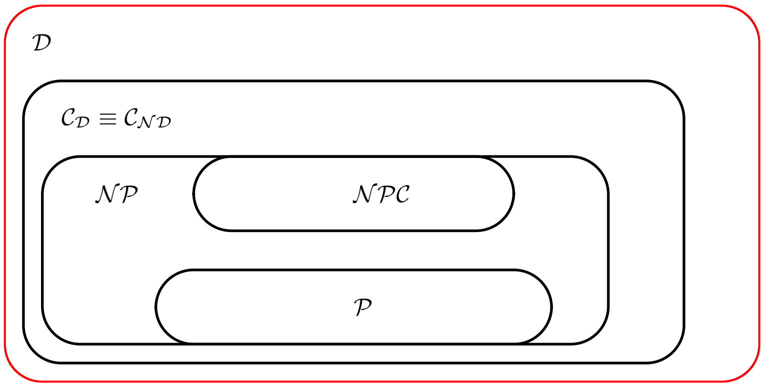
The reducibility relation is transitive: $\boxed{\textbf{If both } X \propto Y \textbf{ and } Y \propto Z \textbf{ then } X \propto Z}$. The reducibility relationship is however not symmetric so it is not an equivalence relation.

9 The class of $\mathcal{NP}$-*Complete* problems

The notion of $\mathcal{NPC}$ can now be introduced formally, with a first definition:

$$\boxed{Z \in \mathcal{NPC} \quad \textbf{if both} \quad \begin{cases} 1: Z \in \mathcal{NP} \\ 2: \textbf{ For all } R \in \mathcal{D}: \textbf{ if } R \in \mathcal{NP} \textbf{ then } R \propto Z \end{cases}} \tag{8}$$

In other words: A problem Z is in $\mathcal{NP}$-*Complete*, if 1) It is in $\mathcal{NP}$ itself, and 2) All other problems that are in $\mathcal{NP}$ also reduce to problem Z, that is, for all $R \in \mathcal{NP}$ it is **TRUE** that $R \propto Z$. As such, a problem that is in $\mathcal{NPC}$ is the most general of all $\mathcal{NP}$-problems.



To stay consistent with other boxed equations, we also present: Thus:

$$\boxed{\boxed{\mathcal{P} \subseteq \mathcal{NP}}} \text{ and } \boxed{\boxed{\mathcal{NPC} \subseteq \mathcal{NP}}} \text{ but the open question is: } \boxed{\boxed{? \quad \mathcal{P} \subseteq \mathcal{NPC} \quad ?}} \tag{9}$$

Even though the definition is short, and its intent is fairly clear: Find the most difficult problem within the $\mathcal{NP}$ problems. BUT: it is hard to identify the very first problem to be a candidate for being inside this $\mathcal{NPC}$, because *all other* $\mathcal{NP}$ problems must reduce to this first problem. It is simply impossible to exhaustively reduce all other $\mathcal{NP}$ problems, since many have been identified but other have not yet been identified. S.A. Cook, however, considered the inner workings of non-deterministic machine when a non-deterministic algorithm operates on it for a polynomial amount of time to solve an instance of any $\mathcal{NP}$ problem. He realized that this working can be represented as Boolean expressions, and that a **yes** is reached if all the variables in these expressions take the correct values. In other words, he realized that the machine always solves a specialized instance of the so called *satisfiability problem (SAT)* (see below).

There is another class of problems that needs to be introduced at this time: A problem H is in $\mathcal{NP}$-*Hard* if

$$\boxed{H \in \mathcal{NPC} \quad \textbf{if only} \quad \begin{cases} 1: ----\text{ (note: } H \text{ need not be in } \mathcal{NP}) \\ 2: \textbf{ For all } R \in \mathcal{D}: \textbf{ if } R \in \mathcal{NP} \textbf{ then } R \propto H \end{cases}} \tag{10}$$

The omission of the first case is intentional: every known algorithm for H, even a non-deterministic one, is not bounded by a polynomial, $T(n) = \Omega(\text{POL}(n))$. Thus H may not be in $\mathcal{NP}$ itself, but surely: 2) All other problems that are in $\mathcal{NP}$ also reduce to problem H, that is, for all $R \in \mathcal{NP}$ it is **TRUE** that $R \propto H$. As such, a problem that is in $\mathcal{NPH}$ is even more general than any $\mathcal{NP}$-problem.

10 The Satisfiability Problem and Cook's Theorem

The satisfiability problem is a decision problem that is solved when the values **true** or **false** can be assigned to Boolean variables $x_1, x_2, \ldots x_n$, so that the value of a Boolean expression in conjuctive normal form (CNF) in these n variables becomes **true**. The CNF (also called product-of-sums) of a Boolean function is the conjunction (**and**'s) of clauses (also called maxterms), where each clause consists of disjunctions (**or**'s) in which each variable or its negation, but not both, appears only once. Thus, if there is a valuation of the variables so that the CNF expression becomes **true**, then *all* clauses must be satisfied. Every Boolean formula which is not a tautology has an equivalent CNF which is unique (except for the ordering of the clauses in the conjunctions, and except for the ordering of the literals inside the clauses. (A literal is either the variable, or its negation.)

Suppose you start with any Boolean expression f; then converting f to its CNF $F(f)$ can be done mechanically. This CNF $F(f)$ is by no means simpler, shorter, or faster to evaluate than the original expression f. The advantage of the CNF form is that there is only one expression in CNF form, whereas there are an infinite number of other Boolean expression also equivalent to f. Thus if two expressions f and g are given, and you wish to determine if these are equivalent to each other, then it may be easier to convert them first to their CNF's and compare the CNF's, rather then trying to reduce one Boolean form to the other using the Boolean operations.

COOK'S THEOREM

$$\boxed{CNF\text{-}SAT \in \mathcal{NPC} \quad \text{in other words: Every } \mathcal{NP}\text{-problem reduces to } SAT} \tag{11}$$

The formal proof is long, but ingenious. It is based on the observation that as soon as a problem X is identified as being in $\mathcal{NP}$, $X \in \mathcal{NP}$, then there exists a non-deterministic algorithm that runs in polynomial time and returns a **yes**. So the only facts known about the problem, is that that there is a sequence of instructions for a non-deterministic machine, whose length is bounded by a polynomial. By describing all initial conditions, program transitions, and program actions as Boolean expressions, a transformation from this instruction sequence to the **CNF-SAT** can be constructed. This effectively simulates the machine while it executes the program to solve X, and this suffices to show that $X \propto$ **CNF-SAT**. For more detailed proofs, see [9, 8, 12].

After Cook's result, the area of $\mathcal{NPC}$-research started in earnest as proving problems to be in $\mathcal{NPC}$ became a lot easier. R.M. Karp showed a diverse number of problems to be in $\mathcal{NPC}$, [6, 7], and M. Garey and D. Johnson gave a complete reference on the area in 1979, [10]. Since then, thousands of problems have been identified as being in $\mathcal{NPC}$.

For any other problem in $\mathcal{NPC}$, the membership proof is quite different and is based on the transitivity of the reducibility relation:

$$\boxed{Z \in \mathcal{NPC} \quad \textbf{if both} \quad \begin{cases} 1 : Z \in \mathcal{NP} \\ 2 : \textbf{There is some } Y \in \mathcal{NPC} \textbf{ and } \; Y \propto Z \end{cases}} \tag{12}$$

If you are able to construct an deterministic algorithm that solves an $\mathcal{NPC}$ problem, then you have effectively solved the open problem:

$$\boxed{\textbf{IF } Z \in \mathcal{NPC} \textbf{ and } \text{you can show that } Z \in \mathcal{P} \quad \textbf{THEN} \quad \mathcal{P} = \mathcal{NP}} \tag{13}$$

It is clear how the open question $\mathcal{P} = \mathcal{NP}$? could be addressed, as the $\mathcal{NPC}$ problems are the best candidates for *not* being in $\mathcal{P}$. Thus:

1. Find as many problems as possible that are all equivalent to SAT.
2. for any single one of these problems, try to show it is in $\mathcal{P}$ (and thus concluding that $\mathcal{P} = \mathcal{NP}$), ór could not possibly be in $\mathcal{P}$, (and thus concluding that $\mathcal{P} \neq \mathcal{NP}$).

11 Additional NPComplete Problems using Problem Reductions

We will show in class (notes to follow next semester (or later this semester)). Please take very good notes.

SAT reduces to $3SAT$

SAT reduces to k-$Clique$

k-$Clique$ reduces to k'-$NodeCover$

k'-$NodeCover$ reduces to $Hamiltonian$-$Cycle$

12 Final Remarks and Acknowledgements

REMARK 19:
There are problems that cannot be affirmed: Take e.g. the Halting Problem, but there are several others known. In fact, there are more problems that cannot be affirmed than there are problems that can be affirmed (an application of Cantor's Diagonalization principle. – these notes will be included next year.)

REMARK 20:
The introduction of $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NPC}$ is only the start of a hierarchy in the world of (decision) problems: $\mathcal{CO}$-$\mathcal{P}$, $\mathcal{CO}$-$\mathcal{NP}$, $\mathcal{CO}$-$\mathcal{NPC}$, $\mathcal{NPH}$, $\mathcal{P}$-$\mathcal{SPACE}$, $\mathcal{LOG}$-$\mathcal{SPACE}$,$\mathcal{P}$-$\mathcal{EXP}$, $\mathcal{NP}$-$\mathcal{EXP}$ and so on. These are not covered in this course.

The question whether $\mathcal{NP} \subseteq \mathcal{P}$ is still an open research question, and many scientists have attempted to solve it. They have not answered the basic question as far as we know, but they have discovered many other interesting things along the way, or found many parallels with other open questions in science. In a way, $\mathcal{NPC}$ theory is only the beginning, as more and more groups of problems are classified into what is called a "polynomial hierarchy" of classes of problems. These are fairly easy to construct by placing the words "For all instances, there exist a ..." in front of a problem already defined. Definitions like $\mathcal{NP} - \mathcal{H}ard$, co-$\mathcal{NP}$ and co-$\mathcal{NPC}$ have been introduced, just like notions of "generable" or "recursively enumerable" are important to show that one can never conclude that a seemingly run-away program has stopped making useful computations (halting problem).

What should you do if the problem you need to solve is $\mathcal{NPC}$? If you are very lucky, it just may be that the particular problem size you have to deal with is not yet in the asymptotic neighborhood. Or, the particular inputs that will be given to you are not the "worst-case" kind, so *your* worst case may be very acceptable. Also, there may be special properties in *your* situation that you can capitalize on to reduce the worst case complexity. Otherwise you might be able to use randomization or probabilistic algorithms to effect acceptable behavior. If these approaches fail, you will have to use approximation algorithms or sub-optimal solutions based on heuristics or based on a limited number of steps in branch-and-bound, pruning, backtracking, and others. These can perhaps be combined with parallel machines for additional speed-up.

The word "reducibility" has the connotation of "simplification", and it implies "cheaper". There are other words that may have been better choices, like "embedded", "more general", "weaker" (in the mathematical sense), "more powerful" (in the sense that it can solve at least two problems), or simply "transformable'" which is used by many authors at the suggestion of D. Knuth. Furthermore, it should be mentioned that there are slightly different definitions of reducibility introduced in the literature.

For a continuously updated catalog for $\mathcal{NP}$ optimization problems, see [13].

Acknowledgements

This section is based on lecture notes we took when taking this course at UNL from XXXX, and from lecture notes we made throughout the many years of teaching senior level and a first year graduate courses on the design and Analysis of Algorithms. As such, the inspiration has come from many different sources, the main ones are listed below in the references. Any omissions are not intentional.

An earlier version appeared as Appendix B in the book:
M. Pióro and D. Medhi:
Routing, Flow, and Capacity Design in Communication and Computer Networks,
Morgan Kaufmann, July 1, 2004. (765 pages).

References

[1] **NOTE:** *these references are quite old and need to be updated. There are many good explanations around on the web, but only few take the* CHOICE-SUCCESS-FAILURE *approach to explain it*

[2] S. A. Cook *The complexity of Theorem Proving Procedures*, Proc. Third Annual ACM Symposium on the Theory of Computing, ACM, New York, 1971, pp 151-158.

[3] S. Even, A. Itai, and A. Shamir, *On the Complexity of Timetable and Multicommodity Flow Problems*, SIAM J. Comput, vol 5, 1976, pp 691-703.

[4] S. Even, *Graph Algorithms*, Computer Science Press, New York, 1979.

[5] S. Skiena *The Algorithm Design Manual*, Springer Verlag, New York, 1997.

[6] R. M. Karp, *Reducibility among Combinatorial Problems*, in: R. Miller and J. Thatcher, eds, *Complexity of Computer Computations*, Plenum, New York, 1972, pp 85-103.

[7] R. M. Karp, *On the Computational Complexity of Combinatorial Problems* , Networks, J. Wiley and Sons, Vol 5, 1975, pp 45-68

[8] Herbert Wilf, *Algorithms and Complexity*, Prentice Hall, New Jersey, 1986.

[9] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms/C++*, Computer Science Press, New York, 1997.

[10] M. Garey and D. Johnson, *Computers and Intractability: A guide to the theory of intractability*, Freeman, San Fransico, 1979.

[11] A. Gibbons, *Algorithmic Graph Theory*, Cambridge university Press, Cambridge, 1985.

[12] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms*, Prentice Hall, New Jersey, 1977.

[13] http://www/nada.kth.se/∼viggo/problemlist/compendium.html

13 Sample Questions

Comment on the following statements (such as: known to be TRUE, known to FALSE, or OPEN QUESTION). Careful: some questions may have been repeated.

1. If a decision problem X is in $\mathcal{NPC}$, then the worst case time complexity of every known deterministic algorithm (i.e. published) is "beyond polynomial", i.e. $T^W(n) = \Omega(\text{POL}(n))$

2. If a decision problem X is in $\mathcal{NPC}$, then there is a non deterministic algorithm known whose worst case time complexity is "polynomial time", i.e. $T^W(n) = \mathcal{O}(\text{POL}(n))$.

3. If a decision problem X is in $\mathcal{NPC}$, and if next year someone discovers an deterministic algorithm G that solves it in polynomial time, i.e $T^W_G(n) = \mathcal{O}(\text{POL}(n))$, then this proves that $\mathcal{P} = \mathcal{NP}$.

4. If a decision problem X is in $\mathcal{NPC}$, and if next year someone discovers and publishes a deterministic algorithm G that solves it in polynomial time, i.e $T^W_G(n) = \mathcal{O}(\text{POL}(n))$, then every $\mathcal{NPC}$ problem can be solved in polynomial time.

5. The Halting Problem is a decision problem that has provably no solution. This means that there will never be an algorithm on a (non-) deterministic machine that affirms it.

6. If a decision problem X is in $\mathcal{P}$, then the worst case time complexity of every known deterministic algorithm A that solves X is $T^W_A(n) = \Omega(\text{POL}(n))$

7. If a decision problem X is in $\mathcal{NPC}$, then the worst case time complexity of every published deterministic algorithm that affirms X is "at least polynomial", i.e. $T^W(n) = \Omega(\text{POL}(n))$.

8. If a decision problem X is in $\mathcal{NPC}$, and if in the future someone publishes a new deterministic algorithm G that affirms it and G runs in polynomial time, i.e $T^W_G(n) = \mathcal{O}(\text{POL}(n))$, then this proves that $\mathcal{P} = \mathcal{NP}$.

9. If a decision problem X is in $\mathcal{NPC}$, and if in the future someone publishes a new deterministic algorithm G that affirms it and G runs in polynomial time, i.e $T^W_G(n) = \mathcal{O}(\text{POL}(n))$, then every $\mathcal{NPC}$ problem can be affirmed (i.e. solved) in polynomial time.

10. If a decision problem X is in $\mathcal{NP}$, and if in the future someone publishes a new non-deterministic algorithm G that affirms it and G runs in polynomial time, i.e $T^W_G(n) = \mathcal{O}(\text{POL}(n))$, then this proves that $\mathcal{P} = \mathcal{NP}$.

11. If a decision problem X is in $\mathcal{NPC}$, and if in the future someone publishes a new non-deterministic algorithm G that affirms it and G runs in polynomial time, i.e $T^W_G(n) = \mathcal{O}(\text{POL}(n))$, then this proves that $\mathcal{P} = \mathcal{NPC}$.

12. If a decision problem X is in $\mathcal{NPC}$, and if in the future someone publishes a new non-deterministic algorithm G that affirms it in polynomial time, i.e $T^W_G(n) = \mathcal{O}(\text{POL}(n))$, then every $\mathcal{NPC}$ problem can be affirmed by a new non-deterministic algorithm in polynomial time.

13. (GS)[1] Suppose there is a decision problem $X \in \mathcal{NP}$ and $X \propto Y$ for every decision problem $Y \in \mathcal{NP}$, then this proves that $Y \in \mathcal{NPC}$

14. (GS) Suppose there is a decision problem $X \in \mathcal{NPC}$ and that $X \propto Y$ for every decision problem $Y \in \mathcal{NP}$, then this proves that $X \in \mathcal{P}$

15. (GS) Suppose there is a decision problem $X \in \mathcal{NP}$ and that $X \propto Y$ for every decision problem $Y \in \mathcal{NP}$, then this proves that $X \in \mathcal{NPC}$

16. (GS) Suppose there are decision problems X, Y, and Z, such that $X \propto Y$ and $Z \propto Y$. This is sufficient to conclude that $X \propto Z$.

17. (GS) Suppose there is a decision problem $Y \in \mathcal{NP}$ and $X \propto Y$ for every decision problem $X \in \mathcal{NP}$, then this proves that $Y \in \mathcal{NPC}$

18. (GS) Suppose there is a decision problem $Y \in \mathcal{NPC}$ and that $X \propto Y$ for every decision problem $X \in \mathcal{NP}$, then this proves that $X \in \mathcal{P}$

19. (GS) Suppose there is a decision problem $Y \in \mathcal{NP}$ and that $X \propto Y$ for every decision problem $X \in \mathcal{NP}$, then this proves that $X \in \mathcal{NPC}$

20. (GS) Suppose there are decision problems V, W, X, Y and Z, and that all following reductions are true: $X \propto Y \propto Z$, and that $W \propto Y \propto Z$, and that $W \propto V$. If you additionally know that $Y \in \mathcal{NPC}$, can you classify all others as being $\in \mathcal{P}$, $\in \mathcal{NP}$, $\in \mathcal{NPC}$, or $\in \mathcal{NPH}$.

[1] GS: Graduate Students