

This lecture is almost identical to lecture 1-2 of COSC1105-7, and has been written by James Harland, and ‘friends’. The following references are largely the standard references for the course COSC 1105-7: Computing Theory.

H. Lewis and C Papadimitriou, *Elements of the Theory of Computation (2nd ed)*, Prentice-Hall, 1998

D Harel, *Algorithmics*, Addison-Wesley, 1992

M Garey and D Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979

Topic 11 Computational Complexity

S2-2003

Computation “Size”

- How do we estimate the cost of a computation?
- What cost is deemed “acceptable”?
- Given that input size varies, how do we “standardise” costs?
- What properties of such costs are important?
- If the cost is too high, what can we do?

All computing resources are finite, no matter how large they may seem:

- processor speed
- memory (RAM, ROM)
- disk, tape, etc.
- screen size

Hence there is *always* some problem which is too large to be solved by a given platform.

For example, the number of operations need to solve the travelling salesman problem for 100 cities is a 161-digit number.

The number of protons in the known universe is a 79-digit number ...

How can the resources needed by a program be measured?

What resources are required by the computation?

The actual resources used by a program depend upon the programming language, CPU speed, available memory, disk speed, operating system, bus traffic, ...

The analysis

- should not depend on a particular implementation
- should not limit available memory or time
- must allow for all possible algorithms

The measurement of complexity is usually described in terms of how the time (or possibly space) requirements increase as the size of the input to the program increases.

The most common measure of complexity is the amount of time the program takes to run.

“*The difference between time and space is that you can re-use space*”.

In general, we perform a *worst-case analysis*; i.e. we work out the maximum number of operations that could occur.

Worst case analysis:

- guarantees termination within a certain number of steps
- provides an upper bound on the resources needed

S2-2003

An average (or even minimum) performance measurement may sometimes be more useful, but in general it makes the analysis significantly more difficult. (what is average? how often do “bad” cases occur? etc.)

Consider an electronic phone directory.

Aristotle
 Archimedes
 DaVinci
 Einstein
 Gallileo
 Newton
 Plato
 Russell
 Tesla
 Turing

Consider the following “pseudo-code” to find a supplied name:

```
i := 1;
while (name <> book[i] and i < n) do
  i := i + 1;
```

This has complexity n , as in the worst case it will

S2-2003

take n comparisons to find an entry (i.e. the last one in the book).

A better method is to use a *binary search* (assuming that the address book is sorted).

Aristotle
 Archimedes
 DaVinci
 Einstein
 Gallileo
 Newton
 Plato
 Russell
 Tesla
 Turing

```
left := 1; right := n;
found := false;
while (not found and left <= right) do
  i := (left + right)/2;
  if (name = book[i]) then found := true
  else if (name > book[i]) then
    left := i+1
  else right := i-1
```

This has complexity $\log_2 n$.

S2-2003

In order to calculate the maximum number of operations that will be done we analyse the algorithm.

Get array[ind-id]
will always take constant time no matter how big the array is for a particular problem.

```
For i = 1 to n
  print i
```

will increase linearly as n increases

```
For i = 1 to n
  for j = 1 to n
    print i - j
```

will have n^2 steps and will increase quadratically

We often only count the most expensive operation involved.

- Sorting programs — comparisons
- Numerical programs — floating point operations
- Graph algorithms — edge traversals

S2-2003

We measure the time complexity of a program by a function. The *rate of growth* of the function is usually what is most important.

n	0	10	100	1000
$20n + 500$	500	700	2,500	20,500
n^2	0	100	10,000	1,000,000
$n^2 + 2n + 5$	5	125	10,205	1,002,005

Note that the lower order terms have progressively less influence as n gets large.

A function f is of *order* g , written $f = O(g)$ if g is an approximation of the rate of growth of f for large n . For example:

- $n^2 + 2n + 5$ is $O(n^2)$
- $20n + 500$ is $O(n)$
- $n^3 + 500n^2 + 6$ is $O(n^3)$
- $2^n + 30n^6$ is $O(2^n)$

$O(1)$	constant
$O(\log_a n)$	logarithmic
$O(n)$	linear
$O(n \log_a n)$	“n log n”
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^r)$	polynomial
$O(a^n)$	exponential
$O(n!)$	factorial

Often anything larger than exponential is referred to simply as exponential (or sometimes hyperexponential).

$O(a^{\sqrt{n}})$ is becoming more important in practice.

n	$\log n$	n^2	2^n	$n!$
5	2	25	32	120
10	3	100	1,024	3,628,800
20	4	400	1,048,576	$\sim 10^{18}$
30	4	900	$\sim 10^9$	$\sim 10^{32}$
40	5	1,600	$\sim 10^{12}$	$\sim 10^{48}$
50	5	2,500	$\sim 10^{15}$	$\sim 10^{64}$
100	6	10,000	$\sim 10^{30}$	$\sim 10^{161}$
200	7	40,000	$\sim 10^{60}$	$\sim 10^{374}$

Consider a problem with factorial complexity. Assume 1 computation per 10^{-12} seconds.

Computing *continuously* from 0 A.D. until now,

$$\begin{aligned}
 \text{time} &= 2002 \frac{\text{yr}}{\text{per}} \times 365.247 \frac{\text{dy}}{\text{yr}} \times 24 \frac{\text{hr}}{\text{dy}} \times 3600 \frac{\text{sec}}{\text{hr}} \times 10^{12} \frac{\text{comp}}{\text{sec}} \\
 &= 63177796282 \text{ sec} \times 10^{12} \frac{\text{comp}}{\text{sec}} \\
 &= 6.3 \times 10^{22} \text{ computations}
 \end{aligned}$$

we would still not have solved the problem for $n = 23$ (!!).

At a rate 1000 times faster, we would have barely finished solving the problem for $n = 25$.

Polynomial time

The complexity of a problem is measured by the *most efficient* solution to it.

For example, a problem with a linear solution and a quadratic solution is linear.

A problem is *decidable in polynomial time* if there is a program which solves the problem with time complexity $O(n^r)$ for some r . The class of all such problems is denoted as \mathcal{P} .

A problem for which there is no efficient solution is said to be *intractable*.

Generally, it is considered that any problem outside \mathcal{P} is intractable.

Note that this is a broad categorisation only;

- sometimes a polynomial-time algorithm is not efficient enough
- sometimes an exponential algorithm will suffice
- sometimes an algorithm is “in-between”, such as $O(2^{\sqrt{n}})$

For example, an $O(n^2)$ algorithm is often sufficiently less efficient than an $O(n \log n)$ algorithm. For $n = 25$, $n^2 = 625$, whereas $n \log n \approx 80$.

Also, the constant factors can make a difference. For example, $\frac{1}{55}2^n < 1000n^2$ for $n < 25$.

Nondeterminism

The reason for the high complexity of many programs is due to the number of alternative solutions. For example, in a complete graph with n vertices, there are $n!$ circuits in the graph. Hence searching all the possibilities takes $O(n!)$ time.

It is often useful to analyse such problems by asking “If some oracle found me a path, how hard would it be to check it?”. In other words, how much of the complexity of the problem is tied up in the search through the alternatives.

One way to express this is through *nondeterminism*.

Essentially, an algorithm is nondeterministic if it makes an unspecified choice at some point.

One way to think of this is: what is the best possible choice that I could make?

E.g. Cost of minimal path from A to B via way-points

1. Choose minimal path.

2. Calculate path cost.
3. Output answer.

Note: This is purely about how to specify the algorithm. It is **not** about reducing the cost of running the algorithm.

In terms of cost, it corresponds to the cost of *checking the answer*.

In the words of a former Computing Theory student: “*Nondeterminism means never having to say that you are wrong*”.

Nondeterministic computations can have complexities which appear much lower than deterministic ones.

Problem: Is n a prime?

Deterministic solution: Test all numbers $2, 3, \dots, n - 1$ as divisors of n .

This has complexity $O(n)$, as we may test all such numbers and find that none of them divide n .

Nondeterministic solution: Pick a number in the range $2, 3, \dots, n - 1$ and test if it is a factor of n .

If it is a composite number, then one of these choices will work. Hence, the cost is that of checking the answer rather than finding one.

Nondeterminism often seems puzzling at first — why introduce “magic” like this?

- simplifies specification of problems and algorithms
- can often be “compiled away” (i.e. replaced with a deterministic procedure)
- localizes costs
- can sometimes help the discovery of more efficient algorithms

If you like, nondeterminism is to determinism as *specification is to computation*.

Nondeterminism	Determinism
Specification	Computation

A problem is *decidable in nondeterministic polynomial time* if there is a nondeterministic program that solves the problem with time

complexity $O(n^r)$ for some r . The class of all such problems is denoted as \mathcal{NP} .

As a deterministic program is a special case of a nondeterministic program, $\mathcal{P} \subseteq \mathcal{NP}$.

What about the other direction, i.e. is $\mathcal{NP} \subseteq \mathcal{P}$? **No one knows for sure!**. It is strongly suspected that $\mathcal{P} \subset \mathcal{NP}$, but there is no proof for this (yet!).

It is possible to convert a nondeterministic program to a deterministic one (see later), but this does not preserve polynomial time complexity (i.e. the deterministic program takes exponential time).

S2-2003

A Problem in NP

A *Boolean variable* is a variable which may have values 0 or 1. A *clause* is a disjunction of variables or their negations. For example, $x \vee \neg y$ and $x \vee y \vee \neg z$ are clauses.

A formula is in *conjunctive normal form* if it has the form $u_1 \wedge u_2 \wedge \dots \wedge u_n$ where each u_i is a clause.

The *satisfiability problem* is to determine if a given formula in conjunctive normal form is satisfied by a truth assignment.

For example, $(x \vee y) \wedge (\neg y \vee \neg z)$ is satisfied by the assignment $x = 1, y = 0, z = 0$. Note that $\neg x \wedge (x \vee y) \wedge (\neg y \vee x)$ is not satisfied by this assignment (nor by any other).

The satisfiability problem can be shown to be in \mathcal{NP} . (very long!).

For n variables, there are 2^n possible truth assignments.

Hence, exhaustive search is exponential, but checking a particular truth assignment is polynomial.

S2-2003

Key Idea:

A problem is in \mathcal{NP} if a proposed solution to it can be *checked* in polynomial time.

A problem is in \mathcal{P} if a solution to it can be *found* in polynomial time.

Hence if $\mathcal{P} = \mathcal{NP}$, finding a solution is not essentially more difficult than checking one. Fame (and fortune?) awaits anyone who can give a polynomial time algorithm for satisfiability (or for various other problems in NP)

S2-2003

More Problems in NP

- 3-satisfiability: the satisfiability problem in which each clause has exactly three literals (i.e. a variable or its negation).
- vertex cover: A *vertex cover* of the nodes of a graph G is a set of nodes VC such that for each arc (u, v) in G , at least one of u and v is in VC . The *vertex cover problem* is to determine if a graph has a vertex cover with k vertices.
- Hamiltonian circuit: A *Hamiltonian circuit* is a path in a graph which visits each node exactly once and returns to its starting point. The *Hamiltonian circuit problem* is to determine if a given graph has a Hamiltonian circuit.
- Travelling Salesman problem; Given a complete graph with weights (i.e. all points connected with a distance on each edge), find the minimal path which visits every node and returns to the start.
- Integer linear programming + many others (see Garey & Johnson)

S2-2003

“Most” interesting and useful problems seem to be NP-complete (or worse!).

Problem reduction

We classify new problems by *reducing* them to known problems. To solve problem X , transform the problem to an input to a known problem Y , and then solve Y .

For example, to solve the Hamiltonian circuit problem, construct a new graph from the old one as follows:

- For each edge in the original graph, add an edge with weight 1 to the new graph
- For all other pairs of vertices in the original graph, add an edge with weight 10 to the new graph

Clearly there is a Hamiltonian circuit in the original graph iff there is a tour of weight $n + 1$ or less in the new graph. Hence we solve the Hamiltonian circuit problem by *reducing* it to the TSP.

S2-2003

S2-2003

Computational Complexity

11-22

NP-Completeness

Remarkably, all of the problems above can be reduced to each other, and, moreover, the transformation can be done in polynomial time.

Hence a polynomial-time solution to one such problem is a polynomial-time solution to **all** of them!

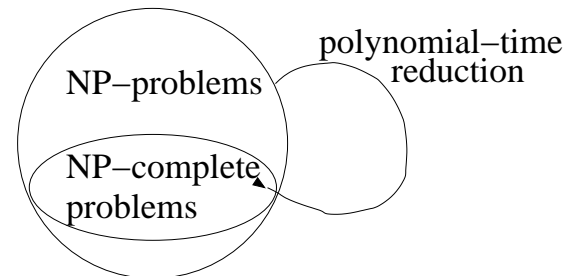
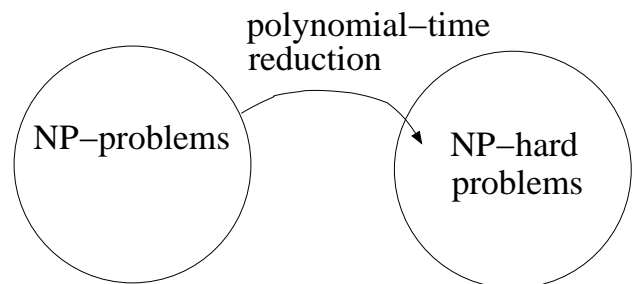
These problems represent completely the whole set of NP problems in that **all** NP problems can be reduced to them in polynomial time. They are called NP-complete.

Not all hard NP problems (i.e. ones we don't have a polynomial algorithm for) are NP-complete.

S2-2003

Computational Complexity

11-23



NP-complete problems

- have solutions which can be *checked* in polynomial time

S2-2003

- cannot necessarily be *found* in polynomial time
- stand or fall together (i.e. either all exponential or all polynomial)
- “straddle” border between tractable and intractable

The first problem shown to be NP-complete was the Boolean satisfiability problem.

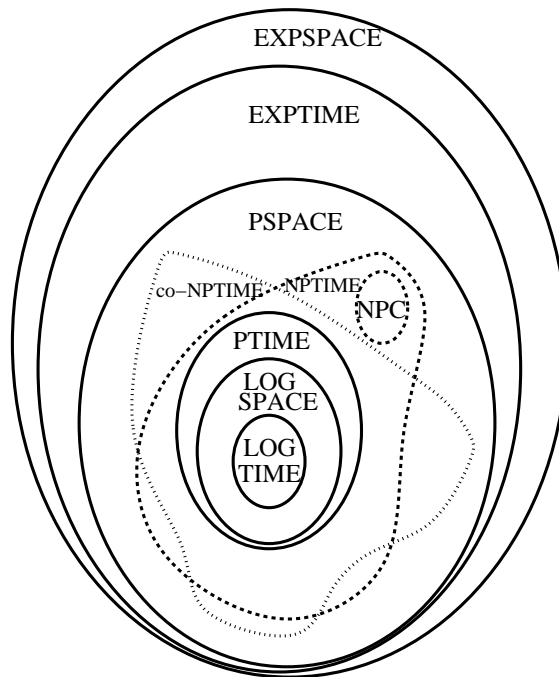
This was done by Cook in 1971, and is considered one of the most important results in computer science. This has a very complex proof (which we will not cover).

Since then, other problems are shown to be NP-complete by finding a reduction of some NP-complete problem to them.

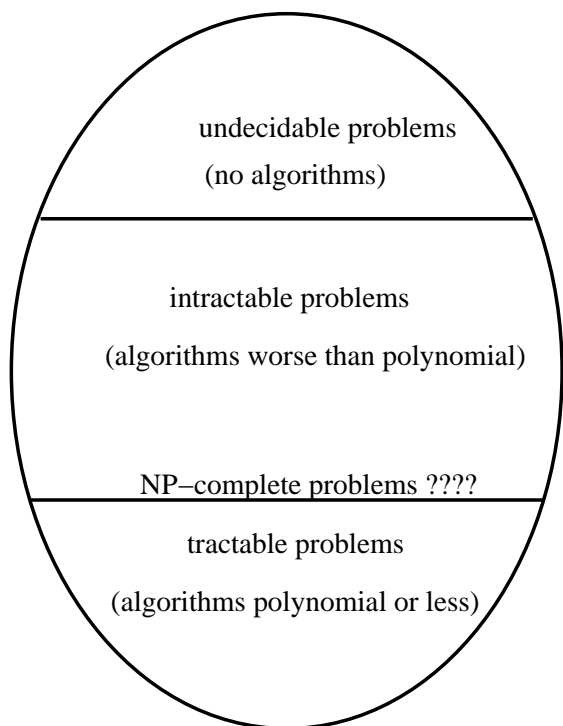
All of the problems in NP mentioned above are NP-complete.

The currently known number of NP-complete problems is well over 1,000.

See the book by Garey and Johnson for more details.



Algorithm Classes



Dealing with NP-Completeness

The traveling salesman problem is a hard problem:

- naive algorithm is $O(n!)$
- best known algorithm is $O(2^{\sqrt{n}})$
- suspected no tractable algorithm exists

How to solve it in practice?

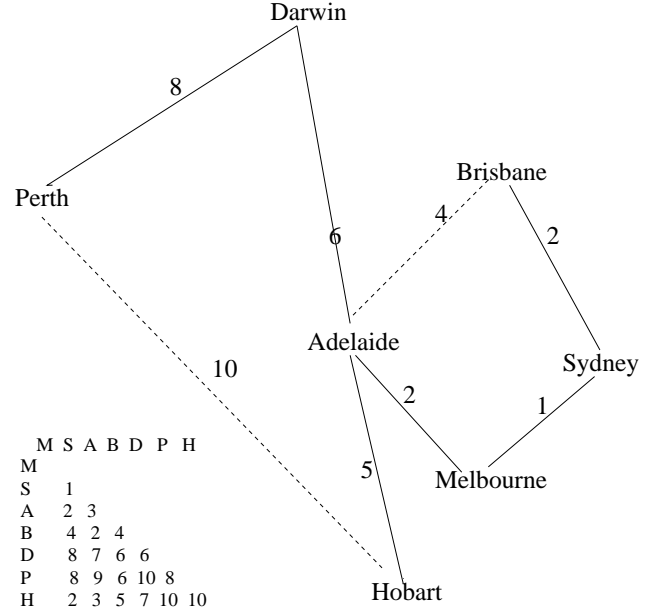
- approximation algorithms: settle for less than optimum
- heuristic algorithms: probably efficient
- randomized algorithms: probably correct

Approximation Algorithms

Relax the constraints — settle for sub-optimal (but not bad) solutions.

- $O(n^2)$ algorithm with solution $\leq 2 \times$ optimum
 - finds minimal spanning tree ($O(n^2)$)
 - preorder traversal ($O(n)$)
 - “forgets” some edges
- $O(n^3)$ algorithm with solution $\leq 1.5 \times$ optimum (more complex than above)

Hence can trade quality for speed.



Output: Melb-Syd-Bris-Adel-Darwin-Perth-Hobart, cost 33

Min. path is 29

Heuristic Algorithms

Try to make “good” choices.

Some special cases of the TSP

- graph is a tree
- triangle inequality: $d(A,B) \leq d(A,C) + d(C,B)$ for all C
- planar graph with Euclidean distance

Heuristics attempt to solve the simple cases first, and only use the full version when these fail.

Heuristics are most useful when worst-cases are rare and average-cases occur often.

Limitations

The classic “big O” concept used here is the worst-case complexity.

This gives a guaranteed upper bound, but can be a little crude.

The *simplex algorithm*, used in linear programming, is exponential, but in practice most cases are not, and this algorithm is widely used in real systems.

Average-case or minimal-case complexity is generally more useful, but more difficult to derive. Some cases are known — for example, it is known that sorting takes *at least* $O(n \log n)$ operations, and also at most $O(n \log n)$ operations (for mergesort).

Find answer with high probability, but not certainty.

Probabilistic algorithms:

- incomplete search
- can measure the maximum error

Other forms of randomized algorithms make random choices in various ways.

- local improvement
- simulated annealing
- genetic algorithms

S2-2003

- Bias parent selection towards good chromosomes
- Swap genetic material between parents
- Mutate some genetic material

until satisfied.

Problems:

- setting parameters
- details of swapping process
- knowing when to stop!

TSP solutions have had some considerable success with such methods.

S2-2003

- general search technique
- often applied when exhaustive search is too hard
- modelled on human genetics
- randomized, but far from totally random

Three main elements:

- Chromosomes to represent the problem (e.g. paths in a graph).
- Fitness function to measure quality of chromosomes.
- Mutation rate for certain random fluctuations.

Start with initial set of chromosomes. Repeat:

- Use fitness function to select “good” chromosomes

S2-2003

Extensions

There is an amazingly intricate hierarchy of complexity classes (parallel classes, probabilistic classes, “oracle” classes, ...) — we have only scratched the surface here.

A new direction is *parametric complexity*, which can be thought of as making more detailed analyses of algorithms in order to explain properties such as “exponential in theory, efficient in practice”.

An interesting notion is *Kolmogorov complexity* — the minimum length encoding of the problem.

For example, the even numbers are a simpler class than the digits of π , as the former have a very simple recogniser, but the latter can only be represented by an infinite string of digits.

There are people working on the $\mathcal{P} = \mathcal{NP}$ question — we may have an answer one day!

S2-2003

Complexity Summary

- Measured by worst-case time on a given length input
- Measure rate of growth of time with respect to length of input
- Tractable \equiv polynomial time or less
- Intractable \geq polynomial time
- Undecidable problems are problems with no algorithm
- NP problems are an interesting subset of (probably) intractable problems.
- To deal with hard problems we can approximate the problem, use heuristics and use probabilistic algorithms.