Complexity

Ian Sanders

Second Semester, 2024

Complexity theory

- We have looked at some non-computable problems.
- We now look at problems that can be solved computationally

Complexity theory

- We have looked at some non-computable problems.
- We now look at problems that can be solved computationally
- Main question – What computational resources are required to solve a given problem?
  - How much time will it take?
  - How much memory will it need?

Complexity theory

- We have looked at some non-computable problems.
- We now look at problems that can be solved computationally
- Main question – What computational resources are required to solve a given problem?
  - How much time will it take?
  - How much memory will it need?
- Complexity theory studies these questions

Complexity theory – Analysis

We can look at

- Worst case analysis
- Best case analysis
- Average case analysis
- Expected case analysis
- Amortised analysis

We won't cover the last two points in this course.

Complexity theory – Bounds

We can look at

- Upper Bounds
- Lower Bounds
- Tight bounds

We start by looking at how to describe these bounds.

Asymptotic notation

We start by determining the *running time* or *time complexity* of the algorithm.

Asymptotic notation

We start by determining the *running time* or *time complexity* of the algorithm.

Recall that...

The TM is the underlying computational model for this theory.

So what we are doing is determining the number of steps executed by a TM to solve the problem.

We can think of each step as executing one quintuple of the TM's program

Asymptotic notation

We start by determining the *running time* or *time complexity* of the algorithm.

Recall that...

The TM is the underlying computational model for this theory.

So what we are doing is determining the number of steps executed by a TM to solve the problem.

We can think of each step as executing one quintuple of the TM's program

In practice, we typically count the number of *basic operations* performed by the algorithm.

Asymptotic notation

We start by determining the *running time* or *time complexity* of the algorithm.

Recall that...

The TM is the underlying computational model for this theory.

So what we are doing is determining the number of steps executed by a TM to solve the problem.

We can think of each step as executing one quintuple of the TM's program

In practice, we typically count the number of *basic operations* performed by the algorithm.

This gives us a function which is dependent on the size of the input – typically n.

We then choose the term that has the highest power of n.

Disregard the coefficient of the term and all lower power terms.

Asymptotic notation

We start by determining the *running time* or *time complexity* of the algorithm.

Recall that...

The TM is the underlying computational model for this theory.

So what we are doing is determining the number of steps executed by a TM to solve the problem.

We can think of each step as executing one quintuple of the TM's program

In practice, we typically count the number of *basic operations* performed by the algorithm.

This gives us a function which is dependent on the size of the input – typically n.

We then choose the term that has the highest power of n.

Disregard the coefficient of the term and all lower power terms.

For example, if our function was $f(n) = 6n^3 + 2n^2 + 20n + 45$ then the function would be asymptotically n^3.

More formally

Let f and g be functions $f, g : \mathrm{N} \rightarrow \mathrm{R}^{+}$. Then we say that $f(n) \in$O$(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.

More formally

Let f and g be functions $f, g : \mathrm{N} \rightarrow \mathrm{R}^+$. Then we say that $f(n) \in$O$(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.

$g(n)$ is an upper bound for $f(n)$

More formally

Let f and g be functions $f, g : \mathrm{N} \to \mathrm{R}^{+}$. Then we say that $f(n) \in \mathrm{O}(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.

$g(n)$ is an upper bound for $f(n)$

$g(n)$ is an asymptotic upper bound for $f(n)$

More formally

Let f and g be functions $f, g : \mathrm{N} \to \mathrm{R}^+$. Then we say that $f(n) \in$O$(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.
$g(n)$ is an upper bound for $f(n)$
$g(n)$ is an asymptotic upper bound for $f(n)$
Example: $f(n) = 5n^3 + 2n^2 + 22n + 6$
$f(n) \in O(n^3)$ ($c = 6$ and $n_0 = 10$).

More formally

Let f and g be functions $f, g : \mathrm{N} \rightarrow \mathrm{R}^{+}$. Then we say that $f(n) \in$O$(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.

$g(n)$ is an upper bound for $f(n)$

$g(n)$ is an asymptotic upper bound for $f(n)$

Example: $f(n) = 5n^3 + 2n^2 + 22n + 6$

$f(n) \in O(n^3)$ ($c = 6$ and $n_0 = 10$).

Note that

$f(n) \in O(n^4)$

(and higher powers of n)

as well

Although, we would normally try to find the smallest power of n that meets the requirement.

More formally

Let f and g be functions $f, g : \mathrm{N} \rightarrow \mathrm{R}^+$. Then we say that $f(n) \in$O$(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.
$g(n)$ is an upper bound for $f(n)$
$g(n)$ is an asymptotic upper bound for $f(n)$
Example: $f(n) = 5n^3 + 2n^2 + 22n + 6$
$f(n) \in O(n^3)$ ($c = 6$ and $n_0 = 10$).
Note that
$f(n) \in O(n^4)$
(and higher powers of n)
as well
Although, we would normally try to find the smallest power of n that meets the requirement.
O is a *tight upper bound*

More formally

Let f and g be functions $f, g : \mathrm{N} \to \mathrm{R}^+$. Then we say that
$f(n) \in$O$(g(n))$ if there exist positive integers c and n_0 such that
for every integer $n \geq n_0$, $f(n) \leq cg(n)$.
$g(n)$ is an upper bound for $f(n)$
$g(n)$ is an asymptotic upper bound for $f(n)$
Example: $f(n) = 5n^3 + 2n^2 + 22n + 6$
$f(n) \in O(n^3)$ ($c = 6$ and $n_0 = 10$).
Note that
$f(n) \in O(n^4)$
(and higher powers of n)
as well
Although, we would normally try to find the smallest power of n
that meets the requirement.
O is a *tight upper bound*
o is a *loose upper bound* ($<$ rather than $\leq$)

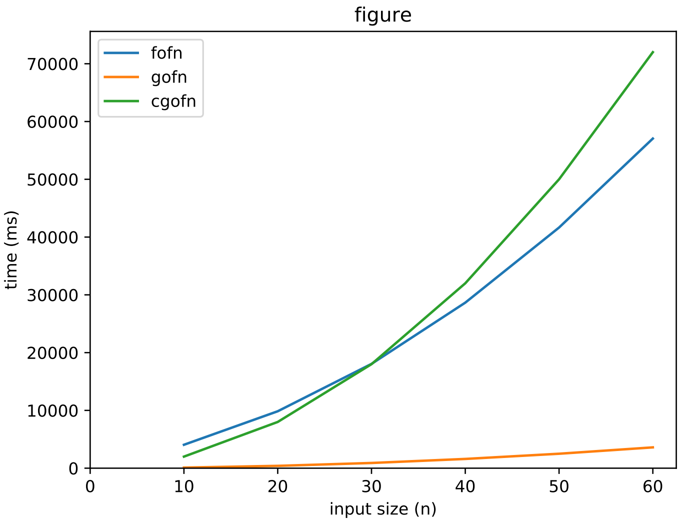


figure
fofn
gofn
cgofn
time (ms)
input size (n)
0
10000
20000
30000
40000
50000
60000
70000
0
10
20
30
40
50
60

Other bounds

Big Ω : tight lower bound (e.g. $\geq$)
Small ω : loose lower bound (e.g. $>$)
Θ : tight upper+lower (e.g. combine Ω, O)

Classes of problems

We have been talking about the performance of *algorithms* (to solve certain problems)

Classes of problems

We have been talking about the performance of *algorithms* (to solve certain problems)

Now we shift the argument to looking at *problems* to determine which class a given problem would fall into...

Problems

Consider the questions below:
What is the shortest path in the travelling salesperson problem?
Does the travelling salesperson problem have a route less that k?

Problems

Consider the questions below:
What is the shortest path in the travelling salesperson problem?
Does the travelling salesperson problem have a route less that k?
The first question has a path as its answer – it is an *optimisation* problem
The second question has either *yes* or *no* as its answer – it is a *decision* problem

Problems

Consider the questions below:

What is the shortest path in the travelling salesperson problem?

Does the travelling salesperson problem have a route less that k?

The first question has a path as its answer – it is an *optimisation* problem

The second question has either *yes* or *no* as its answer – it is a *decision* problem

Any *optimisation problem* has a related decision problem so in some sense they are equivalent.

Problems

Consider the questions below:
What is the shortest path in the travelling salesperson problem?
Does the travelling salesperson problem have a route less that k?
The first question has a path as its answer – it is an *optimisation* problem
The second question has either *yes* or *no* as its answer – it is a *decision* problem
Any *optimisation problem* has a related decision problem so in some sense they are equivalent.
Note: In the discussion below we are typically talking about *decision problems*.

The class P

Polynomial differences in running time are considered small – compare n^2 to n^3.
Exponential differences are considered large

The class P

Polynomial differences in running time are considered small – compare n^2 to n^3.
Exponential differences are considered large
Example: $n = 1000$, then n^3 is 1 billion but 2^n is very large (larger than the number of atoms in the universe).

The class P

Polynomial differences in running time are considered small – compare n^2 to n^3.
Exponential differences are considered large
Example: $n = 1000$, then n^3 is 1 billion but 2^n is very large (larger than the number of atoms in the universe).
Polynomial time algorithms are fast enough for many purposes.
Exponential time algorithms are rarely useful.

Definition for P Class

Definition: P is the class of languages that are decideable in polynomial time on a deterministic single-tape Turing Machine.
$\mathrm{P} = \bigcup_k TIME(n^k)$

Definition for P Class

Definition: P is the class of languages that are decideable in polynomial time on a deterministic single-tape Turing Machine.
$P = \bigcup_k TIME(n^k)$
Important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

Definition for P Class

Definition: P is the class of languages that are decideable in polynomial time on a deterministic single-tape Turing Machine.
$P = \bigcup_k TIME(n^k)$
Important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

1. P is a mathematically robust class, and
2. P is relevant from a practical point of view.

Definition for P Class

Definition: P is the class of languages that are decideable in polynomial time on a deterministic single-tape Turing Machine.
$P = \bigcup_k TIME(n^k)$
Important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

1. P is a mathematically robust class, and
2. P is relevant from a practical point of view.

You have seen examples of these problems.

The class NP

For some problems polynomial time solutions are not known to exist.

The class NP

For some problems polynomial time solutions are not known to exist.
This maybe be because such solutions do not exist

The class NP

For some problems polynomial time solutions are not known to exist.
This maybe be because such solutions do not exist
or
because we just haven't found them yet.

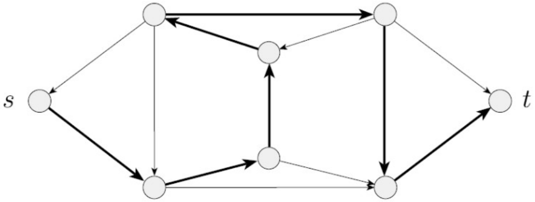
The class NP

For some problems polynomial time solutions are not known to exist.
This maybe be because such solutions do not exist
or
because we just haven't found them yet.
These problems typically have exponential time algorithms.

The Hamiltonian Path problem

A Hamiltonian path in directed graph G is a directed path that visits each node exactly once.

Decision problem: Are two nodes in G connected with a Hamilton path?



The class NP continued

For some problems it is possible to verify the solution (once one has been found) in polynomial time.

The class NP continued

For some problems it is possible to verify the solution (once one has been found) in polynomial time.
NP is the class of problems that have polynomial time verifiers.

The class NP continued

For some problems it is possible to verify the solution (once one has been found) in polynomial time.
NP is the class of problems that have polynomial time verifiers.
NP comes from *nondeterministic polynomial time*
Here a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviours on different runs
So we nondeterministically generate a solution in polynomial time and then check it in polynomial time.

The class NP continued

For some problems it is possible to verify the solution (once one has been found) in polynomial time.

NP is the class of problems that have polynomial time verifiers.

NP comes from *nondeterministic polynomial time*

Here a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviours on different runs

So we nondeterministically generate a solution in polynomial time and then check it in polynomial time.

A language is in NP if it is decided by some nondeterministic polynomial time Turing machine.

The P versus NP Question

Is $P = NP$*?* is one of the greatest unsolved problems in theoretical computer science.

Current thinking is that they are not the same.

That implies that P is a subset of NP.

NP-Completeness

Stephen Cook and Leonid Levin discovered some problems in NP whose individual complexity is related to that of the whole class. Examples: Satisfiability, vertex cover, etc.

NP-Completeness

Stephen Cook and Leonid Levin discovered some problems in NP whose individual complexity is related to that of the whole class.
Examples: Satisfiability, vertex cover, etc.
If we could solve one of these problems in polynomial time then we could solve all of the problems in NP in polynomial time.

NP-Completeness

Stephen Cook and Leonid Levin discovered some problems in NP whose individual complexity is related to that of the whole class.
Examples: Satisfiability, vertex cover, etc.
If we could solve one of these problems in polynomial time then we could solve all of the problems in NP in polynomial time.
Important because

- Theoretical side – results about NP-Complete problems apply to the whole class
- Practical side – proving a new problem is NP-Complete gives us information about the new problem.

Proving NP-Completeness

Reduce, in polynomial time, a known NP-Complete problem to the new problem.

Proving NP-Completeness

Reduce, in polynomial time, a known NP-Complete problem to the new problem.
Must show that any instance of the known NP-Complete problem can be transformed to an instance of the new problem.
And, must show that this can be done by means of a polynomial time function.

Proving NP-Completeness

Reduce, in polynomial time, a known NP-Complete problem to the new problem.

Must show that any instance of the known NP-Complete problem can be transformed to an instance of the new problem.

And, must show that this can be done by means of a polynomial time function.

For example, prove that vertex cover is NP-Complete by a reduction from 3SAT.

Details omitted!

Space Complexity

Space and time are two of the most important considerations when we look at practical solutions to problems.

Space Complexity

Space and time are two of the most important considerations when we look at practical solutions to problems.
Space complexity is handled in a similar way to time complexity.

Space Complexity

Space and time are two of the most important considerations when we look at practical solutions to problems.

Space complexity is handled in a similar way to time complexity.

Underlying model is the Turing Machine – we look at the maximum number of tape cells used for an input of size n

This is analogous to looking at the amount of memory needed to solve a problem on a computer.

Space Complexity

Space and time are two of the most important considerations when we look at practical solutions to problems.
Space complexity is handled in a similar way to time complexity.
Underlying model is the Turing Machine – we look at the maximum number of tape cells used for an input of size n
This is analogous to looking at the amount of memory needed to solve a problem on a computer.
Similar theoretical results apply.
We won't discuss these here.

Conclusion

- Complexity classes allow us to group algorithms or problems independent of the computational model.
- Polynomial vs exponential.
- Polynomial is “easy”.
- P $=$ NP? Probably not...
- Completeness and reduction: solve one efficiently, solve them all!
- Space complexity – a different way of looking at things
- Relationship between time and space
- So, there are many unanswered questions!