Algorithms & Complexity: Lecture 2, Time and space complexity

Sam Barrett

March 9, 2021

1 Upper and lower bounds

A simple set of examples for upper and lower bounds could be:

- **Upper bound:** I can clear my flat in a couple of days at most.
- **Lower bound:** It will take me at least a day to clear my flat.

1.1 Upper bound notation

Note: this notation is not only used for time complexity.

Say that we have two functions: $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.

We say that $f(n)$ is $O(g(n))$ if f is **no bigger** than g up to a constant factor. Or more precisely, if there are numbers c and n_o such that, $\forall n, n \geq n_0$ we have $f(n) \leq c \cdot g(n)$.

Example

$$f(n) \leq 15n^3, \forall n \geq 1000$$

In this situation we can say that $f(n)$ is $O(n^3)$.

We have:

- $c = 15$
- $n_0 = 1000$
- $g(n) = n^3$

We say that $f(n)$ is $o(g(n))$ if f is not as big as g, even up to any constant factor. Or more precisely, if, for any $\epsilon > 0$, there is n_0 such that, $\forall n \geq n_0$ we have $f(n) \leq \epsilon \cdot g(n)$

We can therefore see, if $f(n)$ is $o(g(n))$ then $f(n)$ is always also $O(g(n))$ this can be proven if you take c to be 1.

1.1.1 Examples

Example 1

$5n^2 + 17n + 3$ is $O(n^2)$ and $o(n^3)$ and $O(n^3)$ but **not** $o(n^2)$.

- This is the case as we it is clearly no bigger than $O(n^2)$ (up to a constant factor) as it contains a quadratic term.
- It is *small* compared with n^3 (hence $o(n^3)$) as the highest factor again is n^2.
- It is also $O(n^3)$ as if it is no bigger than $O(n^2)$ it follows that it must also be no bigger than $O(n^3)$.
- We cannot, however, say that it is $o(n^2)$ as it cannot be smaller than n^2 due to it containing a quadratic term.

Example 2

$8n \log n$ is $O(n \log n)$ and $o(n^2)$
We can say this as:

- our term cannot be any bigger than $n \log n$ (up to a constant factor)
- It must be smaller than n^2, due to the nature of logarithms.

1.2 Lower bound notation

- We say that $f(n)$ is $\Omega(g(n))$ when $g(n)$ is $O(f(n))$
  Meaning, there c and n_0 such that, $\forall n \geq n_0$ we have $f(n) \geq c \cdot g(n)$
- We say that $f(n)$ is $\omega(g(n))$ when $g(n)$ is $o(f(n))$
- We say that $f(n)$ is $\Theta(g(n))$ when it is both $O(g(n))$ and $\Omega(g(n))$
  Informally we say this means: "$f(n)$ and $g(n)$ are the same, up to a constant factor"

2 Time complexity

2.1 Running time for a machine M

The running time of a machine M is the time taken from the input state, where x sits on the input tape and the other tapes are blank, to reach the halt state ($q_{\texttt{halt}}$).

For any number n, we define $\texttt{WT}_M(n)$ to be the **worst case** running time for an input of length n. For example,

Input	Running time
00	15
01	23
10	7
11	12

Here $\texttt{WT}_M(2) = 23$. If we were to say that $\texttt{WT}_M(n)$ is $O(n^2)$ we are saying that there are numbers n_0 and C such that, $\forall n \geq n_0$, the running time is $\leq Cn^2$.

2.2 DTIME classes

DTIME(n^2) is a **complexity class**, a complexity class can be thought of as a set of decision problems.

A decision problem, $f : \{0,1\}^* \to \{0,1\}$ is in **DTIME**(n^2) when there is some machine (of any sized alphabet or number of tapes) that decides it (f) and has worst case running time in $O(n^2)$.

2.2.1 Example: palindromes

We can again define our set `PAL` of all palindromic bitstrings with a boolean function $f : \{0,1\}^* \to \{0,1\}$.

Given we have a machine $A-B$ which utilises 3 tapes to decide *palindromicity* and has worst case running time $O(n)$. We can therefore say that `PAL` is in **DTIME**(n).

Can this be improved upon?

No! This can be trivially explained as any solution to palindromicity **must** at least read the input string of length n, therefore there must be **at least** n steps to the computation, leading to a best case running time in $\Omega(n)$.

2.3 Polynomial time

We can define the complexity (super) class of **polynomial time decision problems** as:

$$P \stackrel{\text{def}}{=} \bigcup_{k \geq 1} \textbf{DTIME}(n^k)$$

From this definition, you can see that **any** decision problem in $\{\textbf{DTIME}(n^k)\}_{k=0}^{\infty}$ is also in P

2.3.1 Robustness

Is this definition robust?

- Converting a large alphabet into our default alphabet ($\{\triangleright, \square, 0, 1\}$) only multiplies the running time by a constant factor

- Converting a n tape machine to a 3,2 or 1 tape machine **squares** the running time. This is more significant
- Converting a machine whose tapes are infinite in both directions to a machine whose tapes are infinite in only one direction multiplies the running time by a constant factor
- Converting a machine whose tapes are 2 dimensional to a machine whose tapes are one dimensional **squares the running time**. This is more significant.

In **all** of the cases listed above, the notion of polynomial time that you are left with is the **same**. The same class of decision problems are solvable in polynomial time.

Note: this is true for polynomial time (as defined above) but is not the case for linear or quadratic time

For example, `PAL` can be solved in $O(n)$ on a multitape Turing machine but is $\Theta(n^2)$ on a single tape machine.

2.3.2 Size of input

Another common concern is that our data may be represented as a bitstring in more than one way. However, in practical examples, the representations differ by a **constant** factor leading to polynomial time being the same.

2.4 Exponential time

We can define the complexity class of **exponential time decision problems as**:

$$\mathbf{EXP} \stackrel{\text{def}}{=} \bigcup_{k \geq 1} \mathbf{DTIME}(2^{n^k})$$

Therefore, any decision problem in $\mathbf{DTIME}(2^{5n^{17}})$ is in **EXP** and so on.
Also clearly $\mathbf{P} \subseteq \mathbf{EXP}$

3 Space complexity

Although we often regard time complexity as being the most important, there are many cases in which we need to worry about space complexity as well.

3.1 Space usage of a machine M

The **space usage** for an input x is the number of cells on the **work tapes** that are non-blank at some point during execution.

We ignore blank cells as at any point in computation there are infinitely many of these

For any number n we define $\texttt{WS}_M(n)$ to be the worst case space usage for an input of length n.

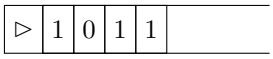
For example:

Input	Space usage
00	5
01	12
10	9
11	9

Here $\texttt{WS}_M(2) = 12$. Saying that $\texttt{WS}_M(n)$ is $O(n^2)$ means that there are numbers n_0 and C such that, $\forall n \geq n_0$, the space usage is $\leq Cn^2$.

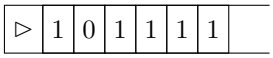
Example execution

Input tape:

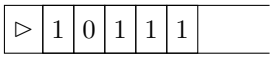


A key point about the calculation of space usage is that we **do not** count the number of non-blank cells on the input tape, only on the work tapes.

Work tape 1:



Work tape 2:



The space usage above would be 11.

3.2 SPACE complexity class

SPACE(n^2) is a complexity class and a decision problem $f : \{0,1\}^* \to \{0,1\}$ is in this space when there is some machine (with any size of alphabet or number of tapes) that decides it (f) and has a worst case space usage in $O(n^2)$

3.3 L and PSPACE

We can now define logarithmic space (**L**) which defines the set of things that can be computed with a machine using a logarithmic number of cells

Note: this relies on the fact that we do not count the number of cells on the input tape. This is because if we were to count the input tape there would be at least n cells used and $n > \log n$, similarly to how we cannot have `PAL` solved in less than linear time.

$$\mathbf{L} \stackrel{\text{def}}{=} \mathbf{SPACE}(\log n)$$

We can also define polynomial space **PSPACE**

$$\mathbf{PSPACE} \stackrel{\text{def}}{=} \bigcup_{k \geq 1} \mathbf{SPACE}(n^k)$$

It is also clear that $\mathbf{L} \subseteq \mathbf{PSPACE}$

3.3.1 Robustness

Is this robust?

Yes, it is in fact more simple than with time.

- Converting a large alphabet into our default alphabet ($\{\triangleright, \square, 0, 1\}$) only multiplies the space usage by a constant factor
- Converting a n tape machine to a 3,2 or 1 tape machine multiplies space usage by a constant factor.
- Converting a machine whose tapes are infinite in both directions to a machine whose tapes are infinite in only one direction multiplies the space usage by a constant factor
- Converting a machine whose tapes are 2 dimensional to a machine whose tapes are one dimensional multiplies space usage by a constant factor.

In all of these cases, logarithmic space does not depend on the model.

3.4 Space vs time

We can show that in all cases, space complexity is less or equal to time complexity.

We can prove by example that $\mathbf{P} \subseteq \mathbf{PSPACE}$:

- Let M be a machine with 5 work tapes that, for any input of length $n \geq 1000$, has a running time of $\leq 18n^3$ steps (a poly-time machine).
- For such an input, the space used is at most $5 + 5 \times 18n^3$

  This is true as 5 cells are non-blank initially and at most 5 more cells per step of execution ($5 \times \texttt{steps}$) $\implies 5 + (5 \times 18n^3)$

We can also show that, in all cases, time is less than or equal to exponentiated space. The following is a proof of $\mathbf{L} \subseteq \mathbf{P}$

- Let M be a machine with 5 work tapes, 74 states 13 symbols and it eventually halts, that, for any input of length $n \geq 1000$ has a space usage of $\leq 18 \log n$ cells

- For such an input, the number of **configurations** is at most

$$74 \times 13^{18 \log n} \times (18 \log n)^5 \times (n + 2)$$

  Where a **configuration** tells us everything about the machine at a given point in execution

  - the state
  - what is written on each work tape
  - where the head is on each work tape
  - where the head is on the input tape

  and

  - 74 is the number of states in which the following apply
  - $13^{18 \log n}$ is the number of possible symbols in each of the maximum number of memory cells
  - $(18 \log n)^5$ represents all the possible head locations over the 5 tapes
  - On the input tape we have $n + 2$ cells in use. This is due to it containing the start symbol $\triangleright$, n bits and a single blank cell.

  We can also see that this number of configurations in bounded by a polynomial as its constituent parts are bounded by polynomials (log etc.).

  The execution time cannot be greater than this because that would mean some configuration is repeated, causing an infinite loop. This is the case as if we reach the same configuration for a second time, there is nothing to prevent it from simply repeating everything it did subsequent to the last time it was in that configuration, thus looping. This cannot be the case as we have assumed our machine M to halt.

Therefore, if the space usage is logarithmic, the running time is polynomial.

The same argument can be made to show that if we have something in polynomial **space** it must be in **exponential** time. To construct this proof simply replace the $\log n$ in the above proof with a polynomial.

4 Nondeterministic time complexity

A simple definition of the complexity class **NP** is

Definition 1 *(**NP**)*
Problems for which checking a solution is easy

There are two methods for formally defining **NP**:

1. using certificates
2. using nondeterministic Turing machines.

4.1 Example: Sudoku

Let SUD be the set of solvable n-Sudoku puzzles, where n refers to the dimension of the grids.

Given a Sudoku puzzle x, a solution **certifies** that $x \in$ SUD

The size of a solution is polynomial in $|x|$ (the length of x). The time taken to check a candidate solution is also polynomial in $|x|$

4.2 Defining NP using certificates

Definition 2 *(***NP***) A language L is said to be in* **NP** *if there is a polynomial-time machine for checking polynomially-sized certificates of L.*

Or, more precisely:

If there is a polynomial p, which gives the size of a candidate certificate) and a polynomial-time machine (for checking a candidate certificate) M such that, $\forall x \in \{0,1\}^*$ (where x is a bitstring representation of a Sudoku puzzle), the following are equivalent:

- $x \in L$
- *There is some bitstring u (a solution to the puzzle) of length $p|x|$ such that, $M\langle x, u\rangle = 1$.*

Here we say that u certifies the fact that $x \in L$

Note above, all text in parenthesis is not a part of the definition

4.3 Nondeterministic Turing machine

A **nondeterministic Turing machine** is similar to a Turing machine except for:

- it has 2 transition functions: δ_0 and δ_1
- besides having a halting state $q_{\texttt{halt}}$ it also has an accepting state $q_{\texttt{accept}}$

It starts in the initial state $q_{\texttt{start}}$, the same as a conventional Turing machine. At each step it *follows* either d_0 or d_1. Once the machine's state is $q_{\texttt{accept}}$ or $q_{\texttt{halt}}$, no further transition takes place.

When we have a nondeterministic Turing machine we need to be more careful when talking about the worst-case time complexity. For example:

Input	Running time
00	15, 7, 3, 9
01	6, **23**
10	7, 11, 5, 11, 8
11	12, 3, 4, 3, 12

Here $\texttt{WT}_M(2) = 23$ and the machine is polynomial-time if WT_M is $O(n^k)$ for some $k \geq 1$

4.4 Defining NP using nondeterministic Turing machines

A language L is in **NP** when there's a polynomial-time nondeterministic machine M such that, for $\forall x \in \{0,1\}^*$, the following are equivalent:

- $x \in L$, x is in the language L.
- When M is executed with input x, there's some sequence of choices that leads to $q_{\texttt{accept}}$

4.4.1 Example: `SUD`

In the case of n-Sudoku, given a Sudoku puzzle x, the nondeterministic Turing machine does the following:

1. begins by copying x to the work tape.
2. Then it non-deterministically fills each blank with a digit. **This stage takes time polynomial in** $|x|$
3. It goes on to check whether the completed grid is valid. **This step (and sub steps) also takes time polynomial in** $|x|$
   (a) If it is, it goes to $q_{\texttt{accept}}$
   (b) if it is not, it goes to state $q_{\texttt{halt}}$.

4.5 Equivalence of definitions

Our two definitions of **NP are** equivalent.

4.6 Is NP = P?

Clearly $\mathbf{P} \subseteq \mathbf{NP}$. This is the case trivially because any deterministic Turing machine is also a nondeterministic Turing machine by simply setting the accepting state to be the same as the halting state, and both transition functions to be the same.

It is an open problem as to whether $\mathbf{P} = \mathbf{NP}$. The currently supported hypothesis is **no**, $\mathbf{P} \neq \mathbf{NP}$. If the answer is *yes*, then there is a polynomial time algorithm for deciding whether an n-Sudoku puzzle is solvable.

It follows that there is a polynomial time algorithm that, given a solvable n-Sudoku puzzle, finds a solution. This is by testing all possible digits for each blank space.

5 Nondeterministic space complexity

Given a nondeterministic Turing machine, what does it mean to be in non-polynomial space complexity?

Let M be a nondeterministic Turing machine. It is in polynomial-space if $\texttt{WS}_M$ is $O(n^k)$ for some $k \geq 1$.

The worst case space complexity is polynomial, therefore, **NPSPACE** is the class of languages that can be decided by a polynomial-space nondeterministic Turing machine. This is the same principal as we saw in our second definition of **NP** in Section 4.4.

The same as $\mathbf{P} \subseteq \mathbf{NP}$, $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$.

5.1 Savitch's theorem

Using a special case of Savitch's theorem, we show $\mathbf{PSPACE} = \mathbf{NPSPACE}$

Suppose that M is a nondeterministic Turing machine, and for an input size n, the space usage is polynomial in n

Then the length of a configuration (as defined earlier) is also polynomial in n.

Let us say that, for $n \geq 1000$, a configuration has length at most $7n^{18}$.

Consider the configuration (directed) graph, which shows all $\leq 2^{7n^{18}}$ different configurations and the transitions between them. Each configuration has at most 2 next configurations.

With this graph, we want to, using a space-efficient algorithm, work out if there's a path from the start configuration to any accepting configuration. If such a path exists we know that the input is accepted.

How do we find this path space-efficiently?

5.1.1 Finding a path space-efficiently

To answer this question, we generalise.

Given nodes s and t and a number k, how much space do we use when deciding whether there is a path of length $\leq 2^k$ from s to t ? We will refer to this result as $D(k)$.

We will argue, by induction, that $D(k) \leq k \times 7n^{18}$.

To do this we check, for each configuration t, whether t is accepting and whether there is a path from the start configuration to t. This requires at most $7n^{18}$ bits to store t, and $D(7n^{18})$ bits to check for the path, i.e. at most $7n^{18} + D(7n^{18})^2$ bits in total. This is polynomial, as is required by *Savitch's theorem.*

All that is left is to finish the inductive proof of $D(k) \leq k \times 7n^{18}$.

Proof 1 *Base case: If $k = 0$, the problem is trivial. Just check if $s = t$.*

Inductive step: To find whether there is a path from s to t of length $\leq 2^{k+1}$, do the following:

1. *For each node z, test whether there is a path from s to z of length $\leq 2^k$ and a path from z to t of length $\leq 2^k$*
2. *By inductive hypothesis, this takes $\leq k \times 7n^{18}$ cells, plus a further*

$7n^{18}$ *cells to store* z*. Totalling* $\leq (k+1) \times 7n^{18}$*.*

Hence, $D(k+1) \leq (k+1) \times 7n^{18}$ *therefore true* $\forall k$