Introduction to Complexity Theory*
Lecture 4: Space Complexity

Notes taken by Leia Passoni and Reuben Sumner

February 4, 1999

Summary: In this lecture we introduce space complexity and discuss how a proper complexity function should behave. We see that properly choosing complexity functions yields as a result well-behaved hierarchies of complexity classes. We also discuss space complexity below logarithm.

1 Complexity Classes

So far two main complexity classes have been considered: $\mathcal{NP}$ and $\mathcal{P}$. We now consider general complexity measures. In order to specify a complexity class, we first have to set the *model of computation* we are going to use, the specific *resource* we want to bound – time or space – and finally the bound itself, that is the *function* with respect to which we want complexity to be measured.

What kind of functions $f : \mathsf{N} \mapsto \mathsf{N}$ should be considered appropriate in order to define "adequate" complexity classes? Such functions should be computable within a certain amount of the resource they bound, and *that* amount has to be a value of the function itself. In fact, choosing a too complicated function as a complexity function could give as a result that the function itself is not computable within the amount of time or space it permits. These functions are not good in order to understand and classify usual computational problems: even though we can use *any* such function in order do formally define its complexity class, strange things can happen between complexity classes if we don't choose these functions properly. This is the reason why we have defined *time constructible functions* when dealing with time complexity. For the same reason we will here define *space constructible functions*.

2 Space Complexity

In space complexity we are concerned with the amount of space that is needed for a computation. The model of computation we will use is a 3-*tape Turing Machine*. We use this model because it is easier to deal with it. We remind that any multi-tape TM can be simulated by an ordinary TM with a loss of efficiency that is only polynomial. For the reminder of this lecture notes, " Turing Machine" will refer to a 3-tape Turing Machine. The 3 tapes are:

1. *input tape*. Read-only

* Lecture Notes for a course given by Oded Goldreich at the Weizmann Institute of Science, Israel.

2. *output tape*. Write-only. Usually considered unidirectional: this assumption is not essential but useful. For decision problems, as considered below, one can omit the output-tape altogether and have the decision in the machine's state.

3. *work tape*. Read and write. Space complexity is measured by the bounds on the machine's position on this tape.

Writing is not allowed on the input tape: this way space is measured only on the worktape. If we allowed writing on the input tape then the length of the input itself should be taken into account when measuring space. Thus we could only measure space complexities which are at least linear. In order to consider also sublinear space bounds we restrict the input tape to be read-only. Define $W_M(x)$ to be the index of the rightmost cell on the worktape scanned by M on input x. Define $S_M(n) = max_{|x|=n} W_M(x)$. For any language L define $\chi_L(x)$ so that if $x \in L$ then $\chi_L(x) = 1$ otherwise $\chi_L(x) = 0$

Definition 1 (Dspace):

$$\text{Dspace}(s(n)) = \{L \,|\exists \text{a Turing machine } M, M(x) = \chi_L(x) \text{and } \forall n \; S_M(n) \leq s(n)\,\}$$

We may multiply $s(\cdot)$ by $\log_2|\Gamma_M|$ where Γ_M is the alphabet used by M. Otherwise, we could always linearly compress the number of space cells using a bigger alphabet. We may also add $\log_2(|x|)$ to $s(\cdot)$, where x is the input. (However, this convention disallow treatment of sub-logarithmic space, and therefore will not be done when discussing such space bounds.) This is done in order to have a correspondence to the number of configurations.

Definition 2 (Configuration) : *A* configuration *of M is an instantaneous representation of the computation carried on by M on a given input x. Therefore if $|x| = n$ a configuration gives information about the following:*

- *state of M* (O(1) bits)
- *contents of the work tape* ($s(n)$ bits)
- *head position in the input tape* ($\log(n)$ bits)
- *head position in the work tape* ($\log(s(n))$ bits)

3 Space Complexities Below Logarithm

Working with sublogarithmic space is not so useful. One may be tempted to think that whatever can be done in $o(\log(n))$ space can also be done in constant space. Formally this would mean

$$\text{Dspace}(o(\log(n))) \subseteq \text{Dspace}(O(1))$$

and since obviously $\text{Dspace}(O(1)) \subseteq \text{Dspace}(o(\log(n)))$, we may also (incorrectly) argue that in fact

$$\text{Dspace}(o(\log(n))) = \text{Dspace}(O(1))$$

This intuition comes from the following imprecise observation: if space is not constant, machine M must determine how much space to use. Determining how much space to use seems to require the machine counting up to at least $|x| = n$ which needs $O(\log(n))$ space. Therefore any M that uses less than $O(\log(n))$ cells, is forced to use constant space. It turns out that this intuition is wrong and the reason is that the language itself can help in deciding how much space to use.

Note: It is known that Dspace($O(1)$) equal the set of regular languages. This fact will be used to prove the following

Theorem 3 Dspace($o(\log(n))$) *is a proper superset of* Dspace($O(1)$).

Proof: We will show that Dspace($o(\log(n))$) $\supseteq$ Dspace($\log\log(n)$) is not contained in Dspace($O(1)$). In fact there is a language L so that $L \in$ Dspace($\log\log(n)$) but $L \notin$ Dspace($O(1)$). For simplicity, we define a language, L, over the alphabet $\{0, 1, \$\}$:

$$L = \left\{ w = 0\cdots0\$0\cdots01\$0\cdots010\$\cdots\$1\cdots1\$ \;\middle|\; \forall k \in \mathsf{N} \;\begin{array}{l}\text{the } l\text{-th substring of } w \text{ delimited by } \$ \text{ has}\\ \text{length } k \text{ and is the binary representation}\\ \text{of the number } l-1, \text{ where } 0 \le l < 2^k\end{array} \right\}$$

It can be easily shown that L is not regular using standard pumping lemma techniques. We then prove that $L \in$ Dspace($\log\log(n)$). Note that $L = \{x_k : k \in \mathsf{N}\}$, where

$$x_k = 0^{k-2}\$0^{k-2}01\$0^{k-2}10\$0^{k-2}11\$\ldots\$1^k\$$$

First consider a simplified case where we only measure space when in fact $x = x_k \in L$, $|x_k| = (k+1)2^k$, but we need to check if $x \in L$. We have to

1. Check the first block is all 0's and the last block is all 1's
2. For any two consecutive intermediate blocks in x_k, check that the second is the binary increment by 1 of the first one.

Step (1) can be done in constant space. In Step (2) we count the number of 1's in the first block, starting from the right delimiter \$ and going left until we reach the first 0. If the number of 1's in the first block is i, we then check that in the second block there are exactly i 0's followed by 1. Then we check the remaining $k - i - 1$ digits in the two consecutive blocks are the same. On input x_k, step 2 can be done in $O(\log(k))$ space, which in terms of $n = |x_k| = (k+1)2^k$, means $O(\log\log(n))$ space.

Handling the case where $x \notin L$ while still using space $O(\log\log(n))$ is slightly trickier. If we only proceeded as above then we might be tricked by an input of the form "$0^n\$$" into using space $O(\log(n))$. We think of x being "parsed" into blocks separated by \$, doing this requires only constant space. We avoid using too much space by making k passes on the input. On the first pass we make sure that the last bit of every block is 0 then 1 then 0 and so on. On the second pass we make sure that the last two bits of every block are 00 then 01 then 10 then 11 and then back to 00 and so on. In general on the i^{th} pass we check that the last i bits of each block form an increasing sequence modulo 2^i. If we ever detect consecutive blocks of different length then we reject. Otherwise, we accept if in some (i.e., i^{th}) pass, the first block is of length i, and the entire sequence is increasing mod 2^i. This multi-pass approach, while requiring more time, is guaranteed never to use too much space. Specifically, on any input x, we use space $O(1 + \log i)$, where $i = O(\log|x|)$ is the index of the last pass performed before termination. ■

Going further on, we can consider Dspace($o(\log\log(n)$) and Dspace($O(1)$). We will show that these two complexity classes are equivalent. The kind of argument used to prove their equivalence extends the one used to prove the following simpler fact.

Theorem 4 *For any* $s(n) : s(n) \geq \log(n)$ $\quad$ Dspace($s(n)$) $\subseteq$ Dtime($2^{o(s(n))}$)

Proof: Fix an input $x : |x| = n$ and a deterministic machine M that accepts x in space $s(n)$. Let be $\mathcal{C}$ the number of possible configurations of M on input x. Then an upper bound for $\mathcal{C}$ is:

$$\mathcal{C} \leq |Q_M| \cdot n \cdot s(n) \cdot 2^{o(s(n))}$$

where $Q_M$is the set of states of M, n is the number of possible locations of the head on the input tape, $s(n)$ is the number of possible locations of the head on the worktape and $2^{o(s)}$ is the number of possible contents in the work tape – the exponent is $o(s)$ because the alphabet is not necessarily binary. We can write $s(n) \cdot 2^{o(s(n))} = 2^{o(s(n))}$ and since s is at least logarithmic , $n \leq 2^{o(s(n))}$. Therefore

$$\mathcal{C} \leq 2^{o(s(n))}$$

M cannot run on input x for a time $t(n) > 2^{s(n)}$. Otherwise, M will go through the same configuration at least twice, entering an infinite loop and never stop. Then necessarily M has to run in time $t(n) \leq 2^{o(s)}$. ■

Theorem 5 Dspace$(o(\log_2 \log_2(n))$ = Dspace$(O(1))$

Proof: Consider a $s(\cdot)$-bounded machine M on the alphabet $\{0, 1\}$.

Claim: given input $x : |x| = n$ such that M accepts x, then M can be on every cell on the input tape at most $k = 2^{s(n)} \cdot s(n) \cdot |Q_M| = O\left(2^{s(n)}\right)$ times. The reason being that if M were to be on the cell more than k times then it would be in the same configuration twice, and thus never terminate.

We define a semi-configuration as a configuration with the position on the input tape replaced by the symbol at the current input tape position. For every location i on the input tape, we consider all possible semi-configurations of M when passing location i. If the sequence of such configurations is $\mathcal{C}^i = \mathcal{C}_1^i, \ldots, \mathcal{C}_r^i$ then by the above claim its length is bounded: $r \leq O\left(2^{s(n)}\right)$. The number of possible different sequences of semi-configurations of M, associated with any position on the input tape, is bounded by

$$\left(2^{s(n)}\right)^{\left(2^{s(n)}\right)} = 2^{2^{O(s(n))}}$$

Since $s(n) = o(\log_2 \log_2 n)$ then $2^{2^{O(s(n))}} = o(n)$ and therefore there exists $n_0 \in \mathsf{N}$ such that $\forall n \geq n_0,\ \ 2^{2^{O(s(n))}} < \frac{n}{3}$. We then show that $\forall n \geq n_0,\ \ s(n) = s(n_0)$. Thus $L \in$ Dspace$(s(n_0))$ = Dspace$(O(1))$ proving the theorem.

Assume to the contrary that there exists an n' such that $s(n') > s(n_0)$. Let $n_1 = \min_{|x|>n_0} \{W_M(x) > s(n_0)\}$ and let $x_1 \in \{0, 1\}^{n_1}$ be such that $W_M(x_1) > s(n_0)$. That is, x_1 is the shortest input on which M uses space more than $s(n_0)$.

The number of sequences of semi-configurations at any position in the input tape is $< \frac{n_1}{3}$. So labelling n_1 positions on the input tape by at most $\frac{n_1}{3}$ sequences means that there must be at least three positions with the same sequence of semi-configurations. Say $x_1 = \alpha a \beta a \gamma a \sigma$. Where each of the positions with symbol a has the same sequence of semi-configurations attached to it.

Claim: The machine produces the same final semi-configuration with either βa or γa eliminated from the input. For the sake of argument consider cutting β leaving $x_1' = \alpha a \gamma a \sigma$. On x_1' the machine proceeds on the input exactly as with x_1 until it first reaches the a. This is the first entry in our sequence of semi-configurations. Locally, M will make the same decision to go left or right on x_1' as it did on x_1 since all information stored in the machine at at the current read head position is identical. If the machine goes left then its computation will proceed identically on x_1' as on x_1 because it still hasn't seen any difference in input and will either terminate or once again come to the first a. On the other hand consider the case of the machine going right. Say this is the ith time at the first a. We now compare the computation of M to what it did following the ith time going past the second a (after the now nonexistent β). Since the semi-configuration is the same in both cases then on input x_1 the machine M also went right on the ith time seeing the second a. The machine proceeded and either terminated or came back for the $i+1$st time to the second a. In either case on input x_1' the machine M is going to do the same thing but now on the first a. Continuing this argument as we proceed through the sequence of semi-configurations (arguing each time that on x_1' we will have the same sequence of semi-configurations) we see that the final semi-configuration on x_1' will be same as for x_1. The case in which γa is eliminated is identical.

Now consider the space usage of M on x_1. Let $x_2 = \alpha a \beta a \sigma$ and $x_3 = \alpha a \gamma a \sigma$. If peak space usage processing x_1 was in αa or σ then $W_M(x_2) = W_M(x_3) = W_M(x_1)$. If peak space usage was in βa then $W_M(x_3) \leq W_M(x_2) = W_M(x_1)$. If peak space usage was in γa then $W_M(x_2) \leq W_M(x_3) = W_M(x_1)$. Choose $x_1' \in \{x_2, x_3\}$ to maximize $W_M(x_1')$. Then $W_M(x_1') = W_M(x_1)$ and $|x_1'| < |x_1|$. This contradicts our assumption that x_1 was a minimal length string that used more than $s(n_0)$ space. Therefore no such x_1 exists.

■

Theorem 3 shows that space below logarithm is not immediately trivial, as we could intuitively think. Theorem 5 shows that on the other hand that this is the case if we consider space complexity functions $s(n) = o(\log\log(n))$.

4 Hierarchy Theorems

As we did for time, we give now

Definition 6 (Space Constructible Function): *A* space constructible function *is a function* $s : \mathsf{N} \mapsto \mathsf{N}$ *for which there exists a machine* M *of space complexity at most* $s(\cdot)$ *such that* $\forall n \quad M(1^n) = s(n)$

For sake of simplicity, we consider only machines which halt on every input. Little generality is lost by this –

Lemma 4.1 *For any space bounded Turing Machine* M *using space* $s(n)$*, where* $s(n)$ *is at least* $\log(n)$ *and space constructible we can construct* $M' \in \mathrm{Dspace}(O(s(n)))$ *such that* $L(M') = L(M)$ *and machine* M' *halts on all inputs.*

Proof: Machine M' first calculates a time bound equal to the number of possible configurations of M which is $2^{s(n)} \cdot s(n) \cdot n \cdot |Q_M|$. This takes space $s(n)$, and same holds for the counter to be maintained in the sequel. Now we simulate the computation of M on input x and check at every step that we have not exceeded the calculated time bound. If the simulated machine halts before reaching its time bound we accept or reject reflecting the decision of the simulated machine. If we reach the time bound before the simulated machine terminates that we are assured that the simulated machine will never terminate, in particular never accept, and we reject the input. ■

Theorem 7 (Space Hierarchy Theorem): *For any space-constructible* $s_2 : \mathsf{N} \mapsto \mathsf{N}$ *and every at least logarithmic function* $s_1 : \mathsf{N} \mapsto \mathsf{N}$ *so that* $s_1(n) = o(s_2(n))$*, the class* $\mathrm{Dspace}(s_1(n))$ *is strictly contained in* $\mathrm{Dspace}(s_2(n))$.

We prove the theorem only for machines which halt on every input. By the above lemma, this does not restrict the result in case s_1 is space-constructible. Alternatively, the argument can be extended to deal also with non-halting machines.
Proof: The idea is to construct a language L in Dspace$(s_2(n))$ such that any machine M using space s_1 will fail recognizing L. We will enumerate all machines running in space s_1 and we will use a diagonalization technique.

- Compute the allowed bound on input x: for instance let it be $\frac{1}{10}s_2(|x|)$.
- Write the language:

$$L = \left\{ x \in \{0,1\}^* \;\middle|\; \begin{array}{l} x \text{ is of the form } \langle M\rangle 01^* \text{ and such that} \\ \text{- } |\langle M\rangle| < \frac{1}{10}s_2(|x|) \\ \text{- on input } x, M \text{ rejects } x \text{ while using at most space } \frac{1}{10}s_2(|x|) \end{array} \right\}$$

  Here $\langle M\rangle$ is a binary encoding of the machine M, so we can see $x \in L$ as a description of M itself.

- Show that $L \in$ Dspace$(s_2(n))$ and $L \notin$ Dspace$(s_1(n))$.

To see that $L \in$ Dspace$(s_2(n))$, we write an algorithm that recognizes L:

<u>On input x:</u>

1. Check if x is of the right form
2. Compute the space bound $S \leftarrow \frac{1}{10}s_2(|x|)$
3. Check the length of $\langle M\rangle$ is correct: $|\langle M\rangle| < \frac{1}{10}s_2(|x|)$.
4. Emulate the computation of machine M on input x. If M exceeds the space bound then $x \notin L$ so we reject.
5. If M rejects x then accept. Else, reject.

The computation in Step (1) can be done in $O(1)$ space. The computation of S in Step (2) can be done in space $s_2(|x|)$ because s_2 is space constructible. Step (3) needs $\log(S)$ space. In Step(4) we have to make sure that (# cells M scans) $\times$ ($\log_2 |\Gamma_M|$) $< S$. Checking that M does not exceed the space bound needs space S. As for the implementation of Step(4), on the work tape we first copy the description $\langle M\rangle$ and then mark a specific area in which we are allowed to operate. Then it is possible to emulate the behavior of M going back and forth on the work tape from $\langle M\rangle$ to the simulated machine's work area, stopping when we are out of space. The algorithm then is running in Dspace$(s_2(n))$.
Note: Since we want to measure space, we are not concerned on how much time is "wasted" going back and forth on the worktape from the description of M to the operative area.

Now we have to see that $L \notin$ Dspace$(s_1(n))$.
We will show that for every machine M of space complexity s_1, $L(M) \neq L$.

There exists $n : s_1(n) < \frac{1}{10}s_2(n)$ since $s_1(n) = o(s_2(n))$. We then consider $M : |\langle M\rangle| < \frac{1}{10}s_2(n)$ and see how M acts on input x of the form $x = \langle M\rangle 01^{n-(|\langle M\rangle|+1)}$ – note that it is always possible to find inputs of the above form for any sufficiently large n. There are two cases:

1. if M accepts x, then (by definition of L) $x \notin L$.
2. if M rejects x, then since $|\langle M \rangle| < \frac{1}{10}s_2(n)$ and $M(x)$ uses at most $s_1(|x|) < \frac{1}{10}s_2(|x|)$ space, $x \in L$.

In either case $L(M) \neq L$. Therefore any M using space s_1 cannot recognize L. ■

Theorem 8 (Time Hierarchy Theorem): *For any time-constructible* $t_2 : \mathsf{N} \mapsto \mathsf{N}$ *and every at least linear function* $t_1 : \mathsf{N} \mapsto \mathsf{N}$ *so that* $\lim_{n\to\infty} \frac{t_1(n)\log(t_2(n))}{t_2(n)} = 0$, *the class* $\mathrm{Dtime}(t_1)$ *is strictly contained in* $\mathrm{Dtime}(t_2)$.

Proof: It is analogous to the previous one used for space. The only difference is in the definition of the language L:

$$L = \left\{ x \in \{0,1\}^* \;\middle|\; \begin{array}{l} x \text{ is of the form } \langle M \rangle 01^* \text{ and such that} \\ \text{- } |\langle M \rangle| < \frac{1}{10}\log(t_2(|x|)) \\ \text{- on input } x, M \text{ rejects } x \text{ while using at most time } \frac{1}{10\log t_2(|x|)}t_2(|x|) \end{array} \right\}$$

Dealing with time, we require $|\langle M \rangle| < \log(t_2(|x|))$. The reason for requiring a small description for M is that we cannot implement Step (4) of the algorithm as it has been done with space: scanning $\langle M \rangle$ and going back and forth from $\langle M \rangle$ to the operative area would blow up time. In order to save time we copy $\langle M \rangle$ in the operative area on the worktape, shifting $\langle M \rangle$ while moving on the worktape to the right. If $|\langle M \rangle| < \log(t_2(|x|))$ it takes then time $\log t_2(|x|)$ to copy $\langle M \rangle$ when needed and time $\log(t_2(|x|))$ to scan it. In Step (4) each step of the simulated machine takes time $O(\log(t_2(|x|)))$ so the total execution time will be

$$\log(t_2(|x|)) \cdot \frac{t_2(|x|)}{10 \cdot \log(t_2(|x|))} = O(t_2(|x|))$$

The logarithmic factor we have to introduce in Step (4) for the simulation of M is thus the reason why in Time Hierarchy Theorem we have to increase the time bound by a logarithmic factor in order to get a bigger complexity class. ■

The Hierarchy Theorems show that increasing the time-space bounding functions by any small amount, gives as a result bigger time-space complexity classes – which is what we intuitively would expect: given more resources, we should be able to recognize more languages.

However, it is also clear that the complexity classes hierarchy is strict only if we use proper time/space bounding functions, namely *time and space constructible* functions. This is not the case if we allow any recursive function for defining complexity classes, as it can be seen in the following theorems.

5 General Complexity Classes

Theorem 9 (Borodin's Gap Theorem): *For any recursive function* $g : \mathsf{N} \mapsto \mathsf{N}$ *with* $g(n) \geq n$, *there exists a recursive function* $s_1 : \mathsf{N} \mapsto \mathsf{N}$ *so that for* $s_2(n) = g(s_1(n))$, *the class* $\mathrm{Dspace}(s_1(n))$ *equals* $\mathrm{Dspace}(s_2(n))$.

Theorem 9 is in a sense the opposite of the Space Hierarchy Theorem: between space bounds $s_1(n)$ and $g(s_1(n))$ there is no increase in computational power. For instance, with $g(n) = n^2$ one gets $g(s_1(n)) = s_1(n)^2$. The idea is to choose $s_1(n)$ that grows very fast and such that even if $g(s_1(n))$ grows faster, no language can be recognized using a space complexity in between.

Theorem 10 (Blum's Speed-up Theorem): *For any recursive function* $g : \mathsf{N} \mapsto \mathsf{N}$ *with* $g(n) \geq n$*, there exists a recursive language* L *so that for any machine* M *deciding* L *in space* $s : \mathsf{N} \mapsto \mathsf{N}$ *there exists a machine* M' *deciding* L *in space* $s' : \mathsf{N} \mapsto \mathsf{N}$ *with* $s'(n) = g^{-1}(s(n))$.

So there exist languages for which we can always choose a better machine M recognizing them.

References

[1] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979

[2] Sipser, M. *Introduction to the Theory of Computation* PWS Publishing Company, 1997

[3] Papadimitriou, C.H., *Computational Complexity* Addison Wesley, 1994