Computational complexity

lecture 2

Other notions of complexity

We are mainly interested in:

- complexity of a language – time and space needed to check that a word belongs to the language

Now we will see other notions of complexity:

- complexity of a word / number – Kolmogorov complexity
- communication complexity

Kolmogorov complexity

Idea: Some numbers (words etc) are easier to remember than other.
They are less complex.
This depends not only on the length of the number.

$$\underbrace{100\ldots0}_{100}$$

$$100^{\overbrace{100^{\cdots^{100}}}^{100}}$$

25839496603316858921

31415926535897932384

Kolmogorov complexity

Idea: Some numbers (words etc) are easier to remember than other.
They are less complex.
This depends not only on the length of the number.

$$\underbrace{100...0}_{100}$$

$$100^{\overbrace{100^{\cdots^{100}}}^{100}}$$

25839496603316858921

20 „random” digits

31415926535897932384

π

$$\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-...=\frac{\pi}{4}$$

Kolmogorov complexity

Idea: complexity of a number = length of its shortest description

Formally: complexity of a number = the size of the smallest Turing machine that outputs this number (when started with empty input)

Kolmogorov complexity

Idea: complexity of a number = length of its shortest description

Formally: complexity of a number = the size of the smallest Turing machine that outputs this number (when started with empty input)

Berry paradox: let n be the smallest number that cannot be defined using ≤100 words (we have just defined it)

Kolmogorov complexity

Idea: complexity of a number = length of its shortest description

Formally: complexity of a number = the size of the smallest Turing machine that outputs this number (when started with empty input)

Berry paradox: let n be the smallest number that cannot be defined using ≤100 words (we have just defined it)

Theorem

A function that maps a number to its complexity is not computable.

Proof

If it is computable, we can also compute the function:

$k \rightarrow$ the smallest number n_k having complexity $\geq k$

(we compute the complexity of consecutive numbers, until we reach a number with complexity $\geq k$)

We see that the complexity of n_k is $\leq C+log(k)$, for some constant C: we output k, and then we apply the function $k \rightarrow n_k$

Thus for every k we have that $k \leq C+log(k)$ – contradiction

Communication complexity

Communication complexity:

- There is a fixed function $f{:}X\times Y\rightarrow Z$ (usually $X=Y=\{0,1\}^n$ $Z=\{0,1\}$).
- Alice knows $x\in X$, Bob knows $y\in Y$.
- How many bits of communication is needed if Alice want to compute $f(x,y)$?

Obviously n bits is always enough, but for some functions it is enough to transfer less bits.

Communication complexity

Communication complexity:

- There is a fixed function $f{:}X\times Y\rightarrow Z$ (usually $X=Y=\{0,1\}^n$ $Z=\{0,1\}$).
- Alice knows $x\in X$, Bob knows $y\in Y$.
- How many bits of communication is needed if Alice want to compute $f(x,y)$?

Obviously n bits is always enough, but for some functions it is enough to transfer less bits.

Example: function „is $x=y$?” requires sending n bits.
Proof: Suppose that it is enough to send n-1 bits. Then there exist two pairs (x,x) and (x',x') for which the communication is identical. Then for the pair (x,x') the communication looks in the same way, so the computed result will be incorrect.

Communication complexity

Communication complexity:

- There is a fixed function $f{:}X\times Y\rightarrow Z$ (usually $X=Y=\{0,1\}^n$ $Z=\{0,1\}$).
- Alice knows $x\in X$, Bob knows $y\in Y$.
- How many bits of communication is needed if Alice want to compute $f(x,y)$?

Obviously n bits is always enough, but for some functions it is enough to transfer less bits.

Example: function „is $x=y$?” requires sending n bits.
Proof: Suppose that it is enough to send n-1 bits. Then there exist two pairs (x,x) and (x',x') for which the communication is identical. Then for the pair (x,x') the communication looks in the same way, so the computed result will be incorrect.

Lower bounds for the communication complexity for appropriate functions were used to prove some lower bounds for complexity of some problems, e.g., for streaming algorithms.
See also: problem 1.5.3 – a single-tape machine recognizing the language of palindromes requires time $\Omega(n^2)$

Sipser's theorem

Theorem. Consider a machine M working in space $S(n)$, but not necessarily having the halting property.
Then there exists a machine M' such that:

- $L(M')=L(M)$
- M' works in space $S(n)$
- M' halts on every input

[now we come back to complexity of languages]

Sipser's theorem

Theorem. Consider a machine M working in space $S(n)$, but not necessarily having the halting property.
Then there exists a machine M' such that:

- $L(M')=L(M)$
- M' works in space $S(n)$
- M' halts on every input

Thus: in the following definition

A language $L\subseteq\Sigma^*$ is *recognizable in space* $S(n)$ if there exists a multitape machine ~~that halts on every input~~, accepts L, and works in space $S(n)$.

this condition was redundant

Sipser's theorem

Theorem. Consider a machine M working in space $S(n)$, but not necessarily having the halting property.
Then there exists a machine M' such that:

- $L(M')=L(M)$
- ~~M' works in space $S(n)$~~
- M' halts on every input

Proof

Approach 1: (in which the resulting M' uses a lot of space)

Key observation: in an accepting run no configuration repeats.

- after every move we copy the current configuration to an additional working tape,
- additionally we check whether the current configuration equals to some configuration saved earlier
- a configuration has repeated ⇒ a loop ⇒ we reject

Sipser's theorem

Theorem. Consider a machine M working in space $S(n)$, but not necessarily having the halting property.
Then there exists a machine M' such that:

- $L(M')=L(M)$
- M' works in space $S(n)$
- M' halts on every input

Proof

Approach 2: a counter of moves:

- an accepting run has at most $c^{S(n)}$ steps, whenever $S(n)\geq log(n)$
- we can count up to $c^{S(n)}$ using a counter of size $S(n)$
- thus we count: we increment the counter by 1 after every step
- the counter overflows ⇒ we reject

Sipser's theorem

Approach 2: a counter of moves:

- an accepting run has at most $c^{S(n)}$ steps, whenever $S(n) \geq log(n)$
- we can count up to $c^{S(n)}$ using a counter of size $S(n)$
- thus we count: we increment the counter by 1 after every step
- the counter overflows ⇒ we reject

Problems:

- We get a machine M, but maybe we cannot compute the function $S(n)$.
  [Actually, we can – this follows from approach 3]

Sipser's theorem

Approach 2: a counter of moves:

- an accepting run has at most $c^{S(n)}$ steps, whenever $S(n) \geq log(n)$
- we can count up to $c^{S(n)}$ using a counter of size $S(n)$
- thus we count: we increment the counter by 1 after every step
- the counter overflows ⇒ we reject

Problems:

- We get a machine M, but maybe we cannot compute the function $S(n)$.
  [Actually, we can – this follows from approach 3]
  This is only an ostensible problem: we know that such a function $S(n)$ exists, so there exists a machine which at the very beginning reserves a counter of this size. Maybe we cannot compute this machine out of M, but the theorem only says that „there exists M' ”
- But does such a machine really exist? The function $S(n)$ has to be space constructible (we will see more on this topic soon)

Sipser's theorem

Approach 2: a counter of moves:

- an accepting run has at most $c^{S(n)}$ steps, whenever $S(n)\geq log(n)$
- we can count up to $c^{S(n)}$ using a counter of size $S(n)$
- thus we count: we increment the counter by 1 after every step
- the counter overflows ⇒ we reject

Problems:

- We get a machine M, but maybe we cannot compute the function $S(n)$.
  [Actually, we can – this follows from approach 3]
  This is only an ostensible problem: we know that such a function $S(n)$ exists, so there exists a machine which at the very beginning reserves a counter of this size. Maybe we cannot compute this machine out of M, but the theorem only says that „there exists M' "
- But does such a machine really exist? The function $S(n)$ has to be space constructible (we will see more on this topic soon)
  The requirement that $S(n)$ is space constructible can be avoided:
  M' does not reverse the whole counter at the very beginning, but it increases it always when M visits a new memory cell – always it holds:
  counter length ≥ log(the number of configurations under the current memory usage).
  Such a counter is sufficient.
- This construction works only when $S(n)\geq log(n)$

Sipser's theorem

Approach 2: summing up – construction of M':

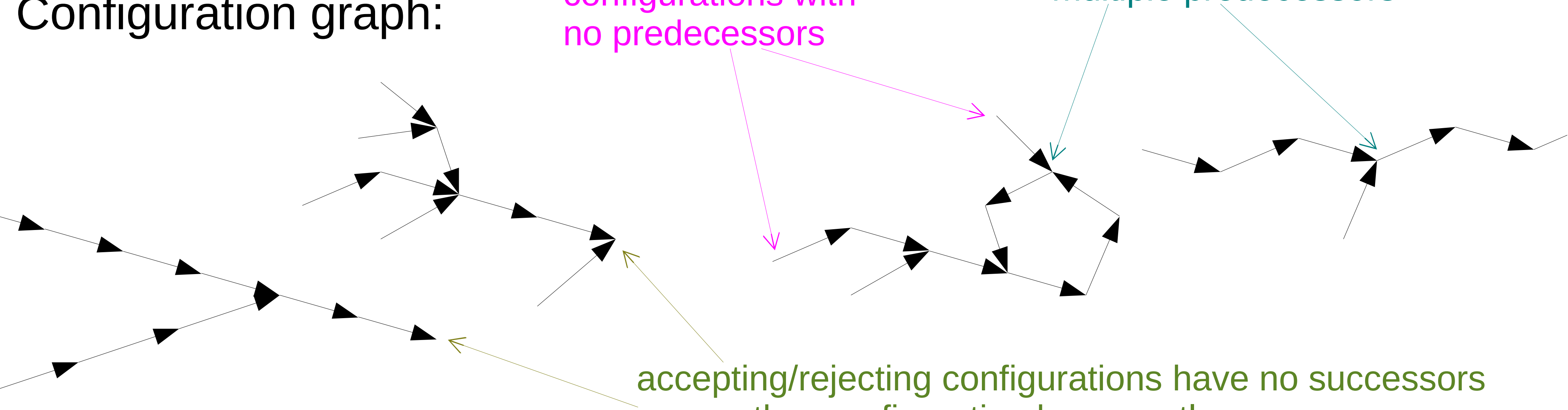
- suppose that M has at most $n \cdot c^{S+1}$ configurations using S cells of memory
- M' works as M, but additionally there is a counter on a separate tape
- at the very beginning M' creates this counter – its value is 0, and its length is $\lceil log(nc) \rceil$
- this counter is increased after every “real” step of M
- when M enters a cell with $\perp$, the counter length is increased by $\lceil log(c) \rceil$ (constant)
- when counter overflows, M' rejects
- this construction uses space $O(S(n)+log(n))$

Sipser's theorem (✶)

Approach 3 [Sipser]: explore the configuration graph going back from accepting configurations

- good: the problem of cycles disappear – there are no cycles at all
- bad: there are infinitely many accepting configurations,
  a configuration may have multiple predecessors,
  there are infinite paths while going back

Configuration graph:



configurations with no predecessors

configurations with multiple predecessors

accepting/rejecting configurations have no successors
every other configuration has exactly one successor

(✶) - Some slides will be marked with this sign. They contain more complicated proofs. If you get lost, this is not a very big problem.

Sipser's theorem (✶)

Approach 3 [Sipser]: explore the configuration graph going back from accepting configurations

- good: the problem of cycles disappear – there are no cycles at all
- bad: there are infinitely many accepting configurations,
  a configuration may have multiple predecessors,
  there are infinite paths while going back

Example: transition $q_1,\perp \rightarrow q_1,\perp,L$

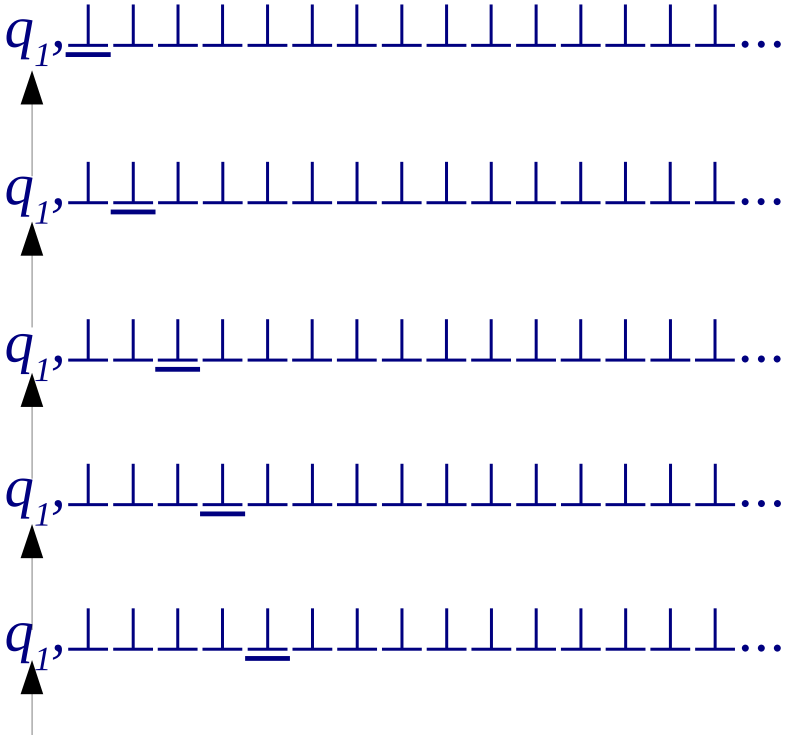
$$\begin{tikzcd}
q_1,\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots \\
q_1,\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots \arrow[u] \\
q_1,\perp\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots \arrow[u] \\
q_1,\perp\perp\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots \arrow[u] \\
q_1,\perp\perp\perp\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots \arrow[u] \\
{} \arrow[u]
\end{tikzcd}$$

Sipser's theorem (✶)

Approach 3 [Sipser]: explore the configuration graph going back from accepting configurations

- good: the problem of cycles disappear – there are no cycles at all
- bad: there are infinitely many accepting configurations,
  a configuration may have multiple predecessors,
  there are infinite paths while going back

Example: transition $q_1,\perp \to q_1,\perp,L$



$q_1,\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots$

$q_1,\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots$

$q_1,\perp\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots$

$q_1,\perp\perp\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots$

$q_1,\perp\perp\perp\perp\underline{\perp}\perp\perp\perp\perp\perp\perp\perp\perp\perp\perp\ldots$

Antidote: Forbid this!

- Assume w.l.o.g. that M never writes $\perp$
- Consider only configurations with no $\perp$ to the left of the head

Sipser's theorem (✶)

Approach 3 [Sipser]: explore the configuration graph going back from accepting configurations

Assumptions:

- M never writes $\perp$
- We consider only configurations with no $\perp$ to the left of the head

Then:

- good: the problem of cycles disappear – there are no cycles at all,
  there is a function: configuration → memory usage,
  memory usage never decreases (while going back: never increases),
  no infinite paths while going back
- bad: there are infinitely many accepting configurations,
  a configuration may have multiple predecessors,
  ~~there are infinite paths while going back~~

Recall that memory usage = number of visited cells.
If M could write $\perp$, seeing only a current configuration we don't know how many cells were already visited.

Sipser's theorem (✶)

Approach 3 [Sipser]: explore the configuration graph going back from accepting configurations

- good: the problem of cycles disappear – there are no cycles at all, there is a function: configuration → memory usage, memory usage never decreases (while going back: never increases), no infinite paths while going back
- bad: there are infinitely many accepting configurations, a configuration may have multiple predecessors

Procedure *Search(C)*: Starting from a configuration C perform the DFS on the configuration graph, looking for the initial configuration.

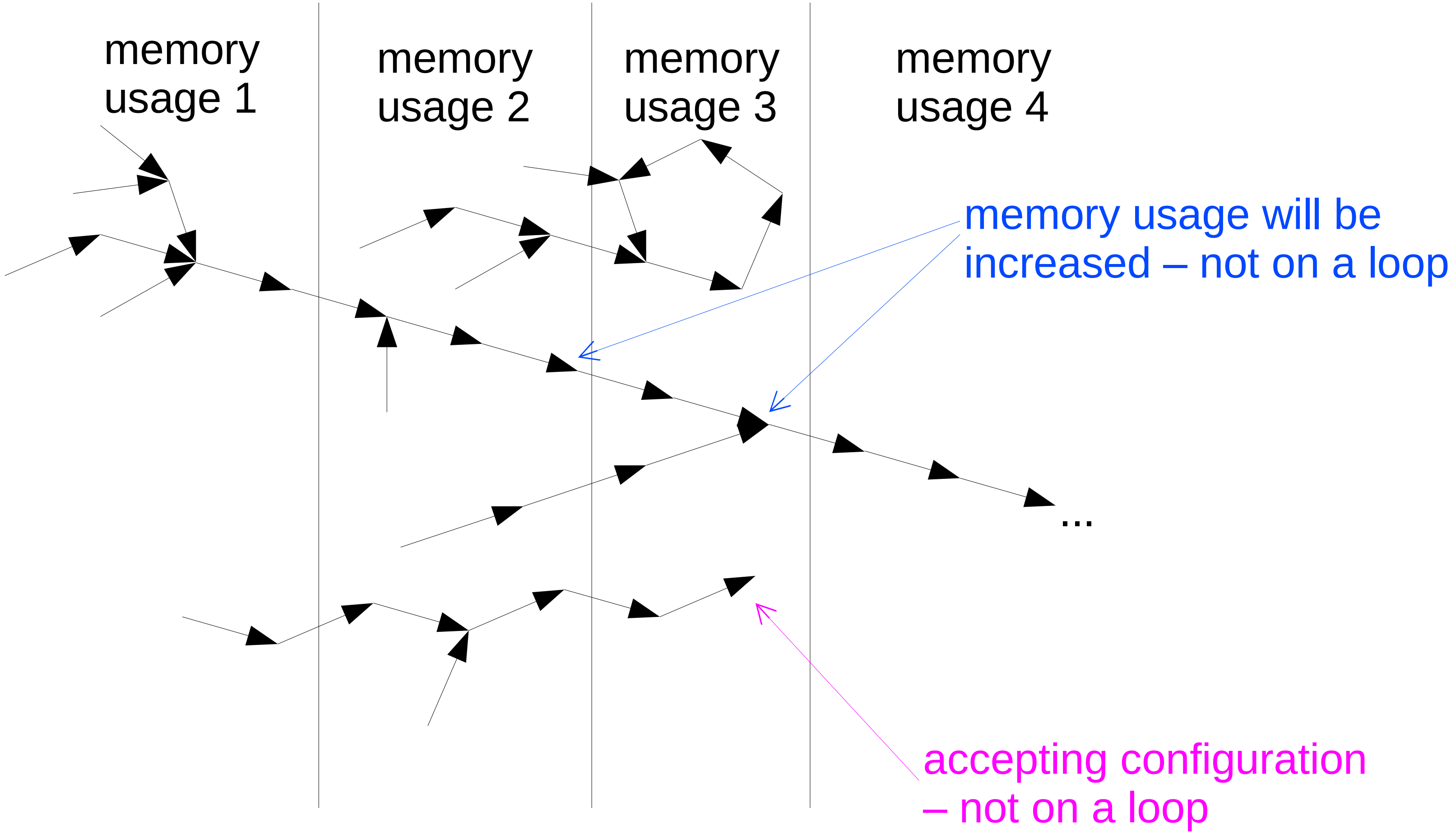
Search(C) works in space k.
If k memory cells are occupied in C, and either C is accepting, or the next step from C increases memory usage, then *Search(C)* halts.

How to perform this DFS in space k? We can only remember the current configuration (OK, as we are in a tree). Additionally we remember whether we have came from the parent in the tree, or from a child; in the latter case also from which child.

Sipser's theorem (✶)

Procedure *Search(C)*: Starting from a configuration C perform the DFS on the configuration graph, looking for the initial configuration.

Search(C) works in space k.
If k memory cells are occupied in C, and either C is accepting, or the next step from C increases memory usage, then *Search(C)* halts.



memory usage will be increased – not on a loop

accepting configuration – not on a loop

Sipser's theorem (✶)

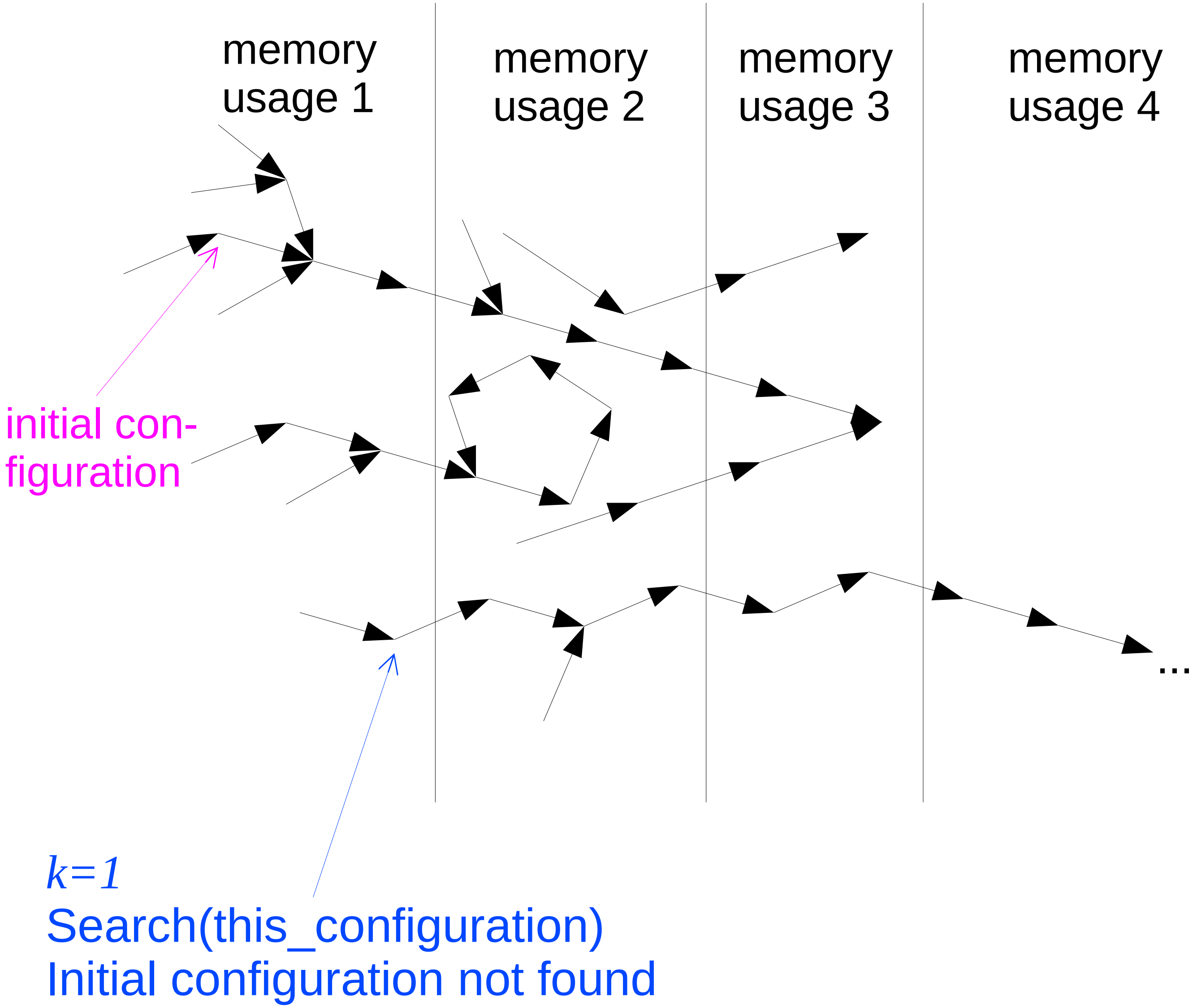
Procedure *Search(C)*: Starting from a configuration C perform the DFS on the configuration graph, looking for the initial configuration.

Search(C) works in space k.
If k memory cells are occupied in C, and either C is accepting, or the next step from C increases memory usage, then *Search(C)* halts.

The algorithm simulating M back:

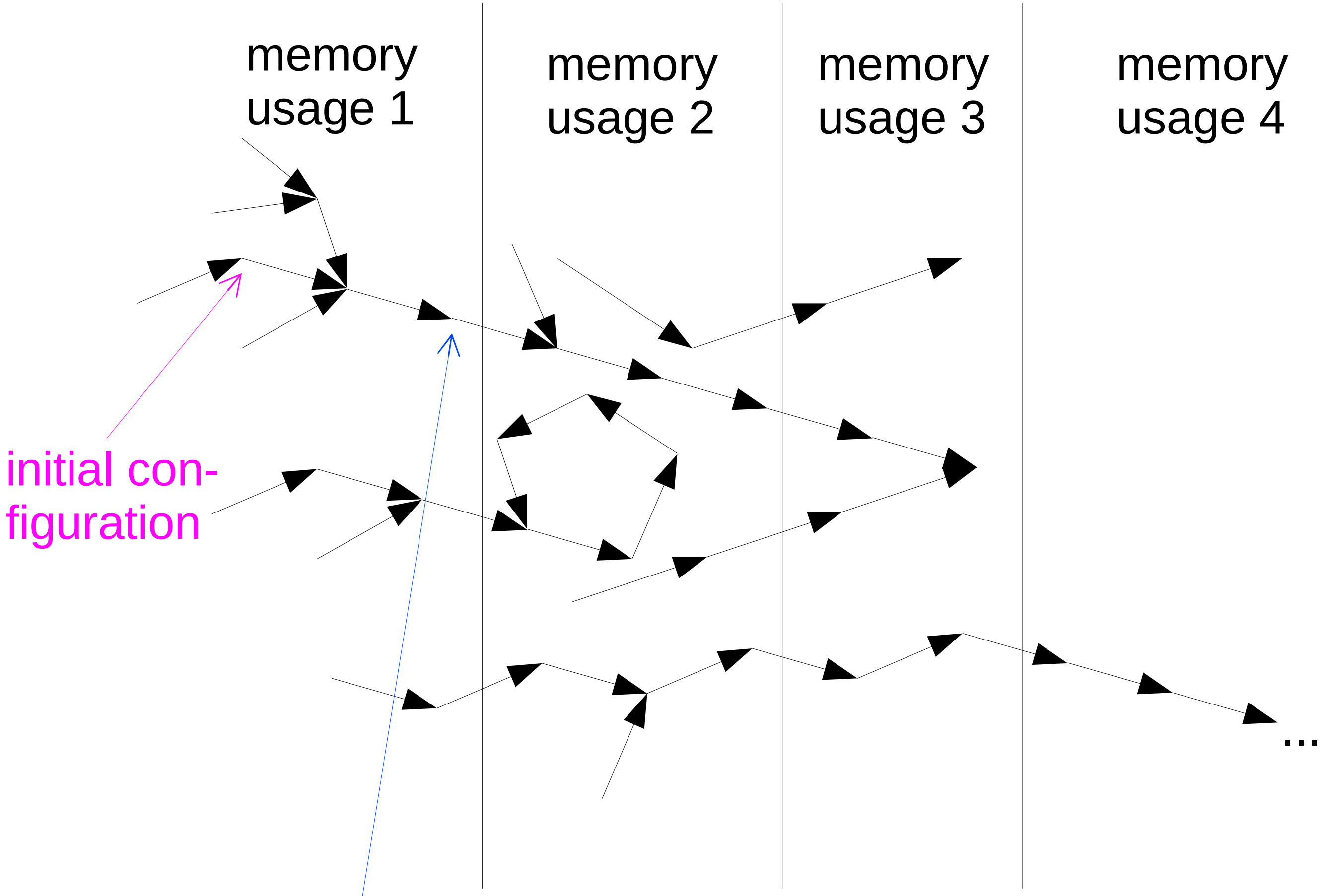
- We assume that M has only one working tape, and never writes $\perp$.
  (can be done without increasing memory usage)
- For consecutive k perform the following steps:
  ➔ Check all configurations using k memory cells
  ➔ If the next step from C increses memory usage, call *Search(C)* and check whether C can be reached from an initial configuration
  If yes, increase k, and repeat the same.
- After this loop we know that M uses exactly k memory cells on the input word. It remains to check whether it accepts.
- To this end, we call *Search(C)* from every accepting configuration using at most k memory cells.

Sipser's theorem (⋆)



k=1
Search(this_configuration)
Initial configuration not found

Sipser's theorem (⋆)



memory usage 1

memory usage 2

memory usage 3

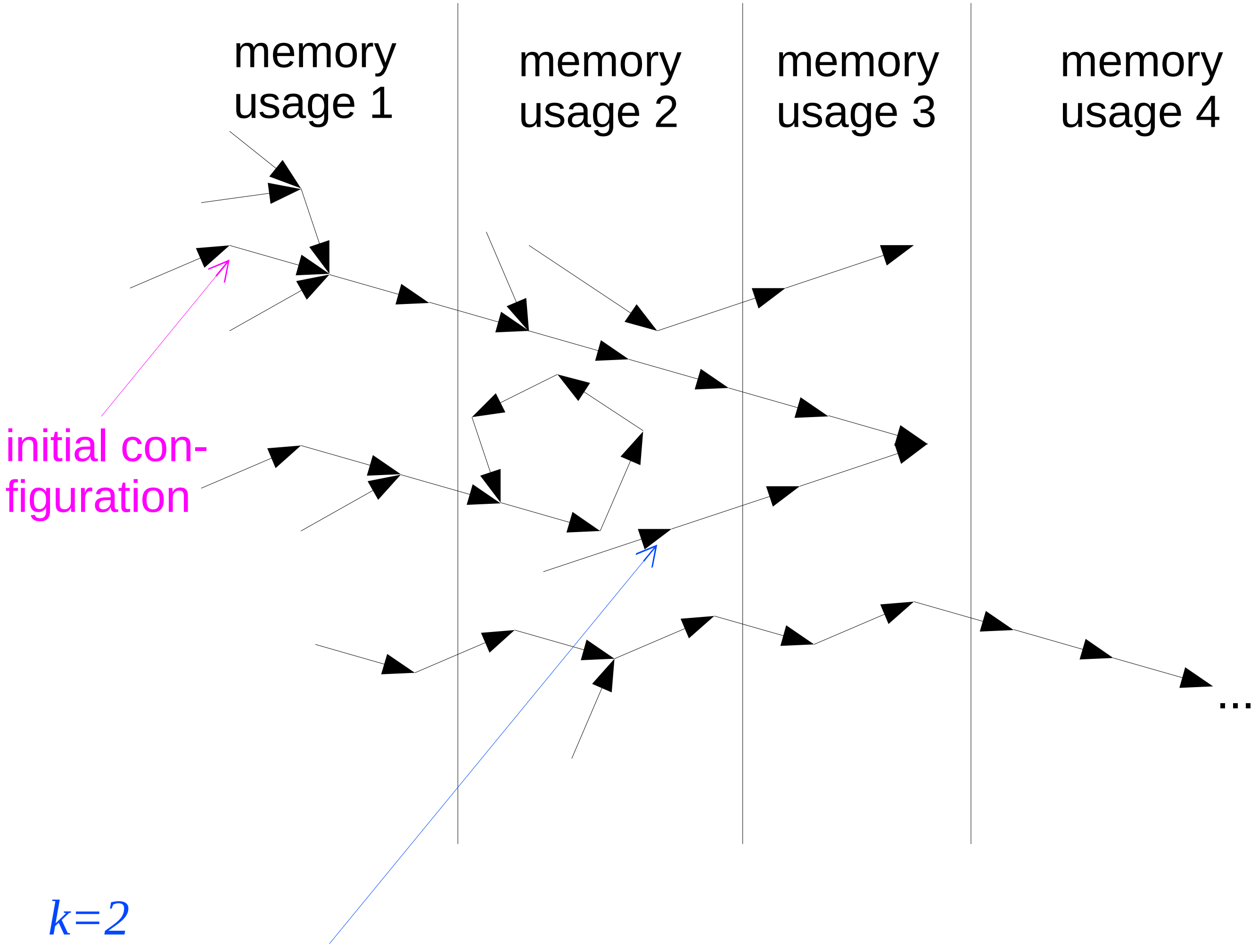
memory usage 4

initial configuration

...

k=1
Search(this_configuration)
Initial configuration found;
Increase *k*

Sipser's theorem (✶)



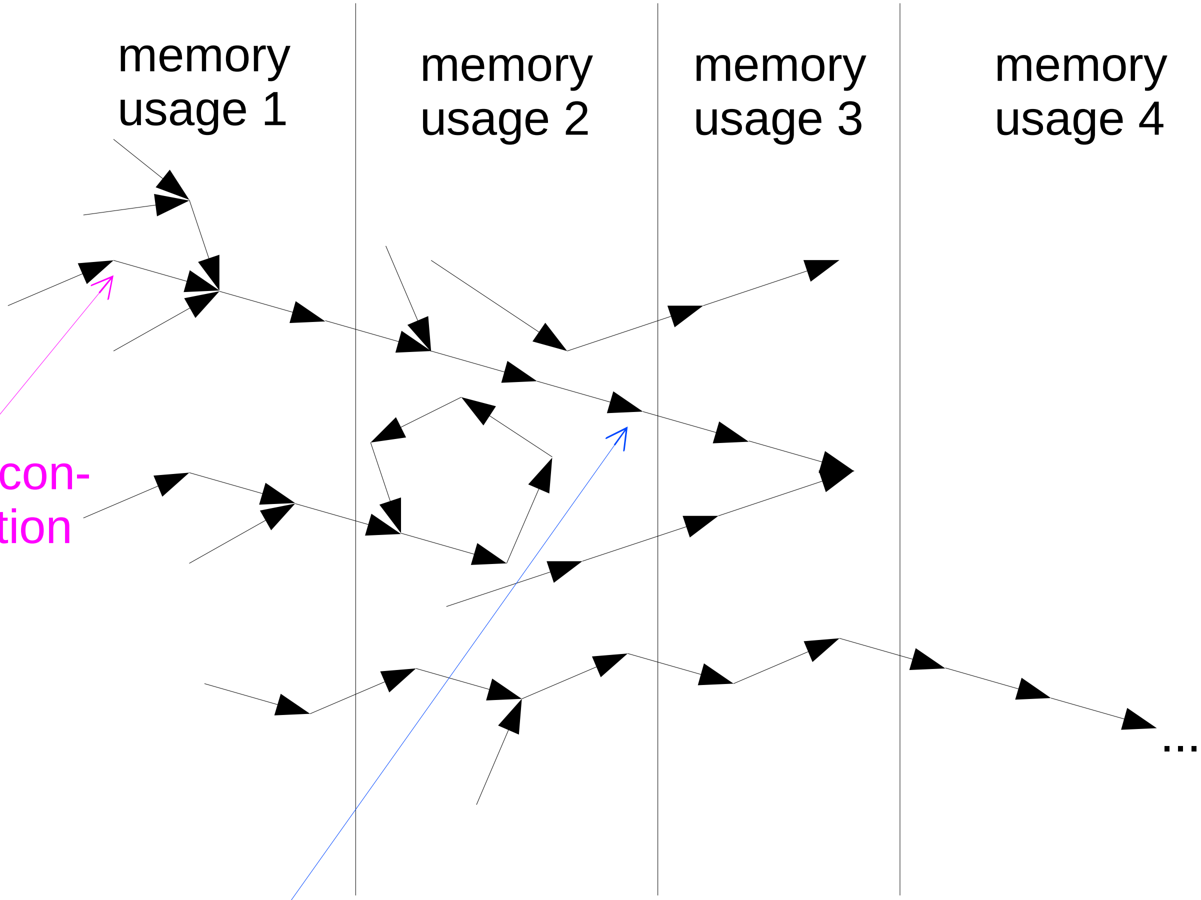
initial configuration

k=2

Search(this_configuration)

Initial configuration not found

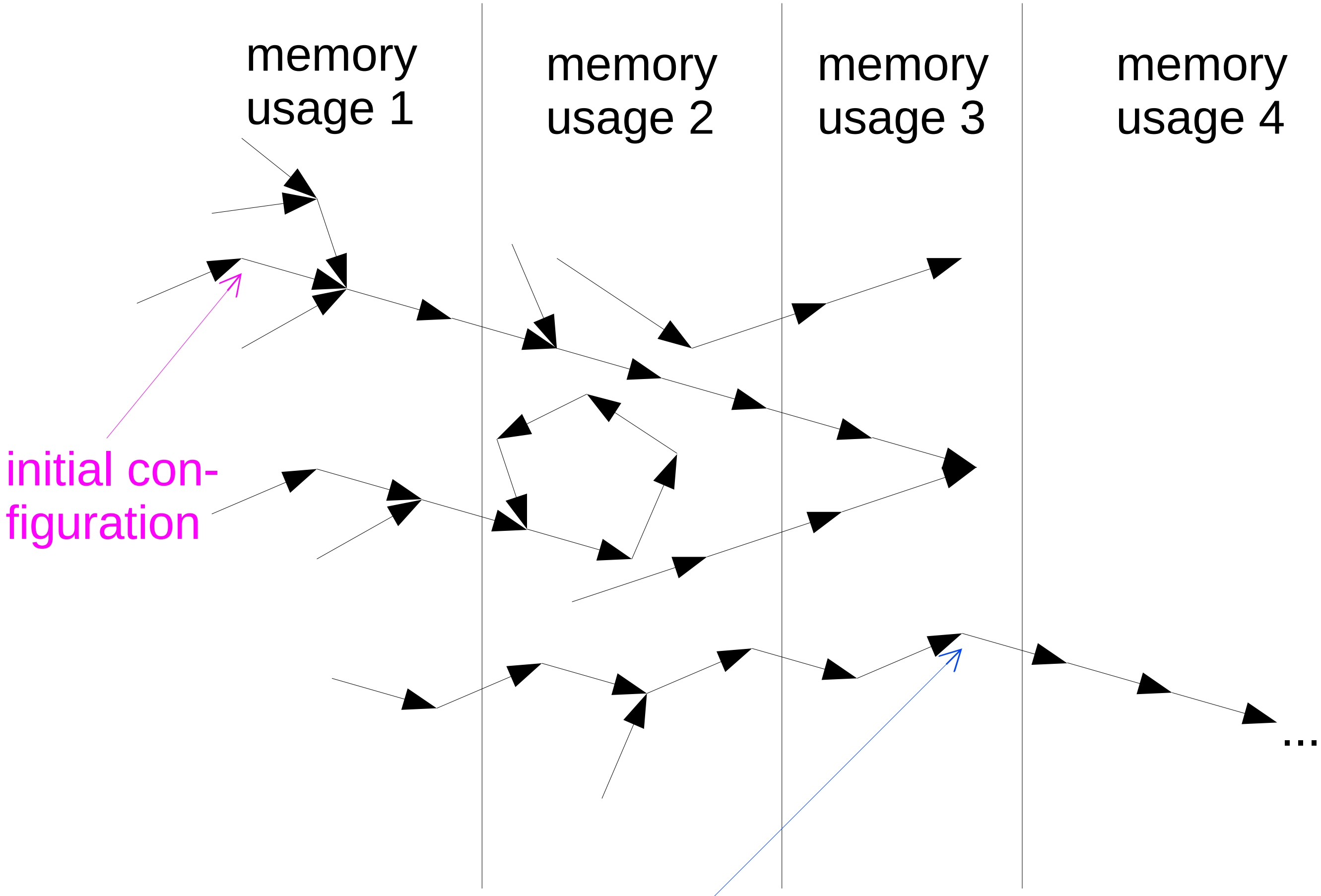
Sipser's theorem (⋆)



initial con-
figuration

k=2
Search(this_configuration)
Initial configuration found;
Increase *k*

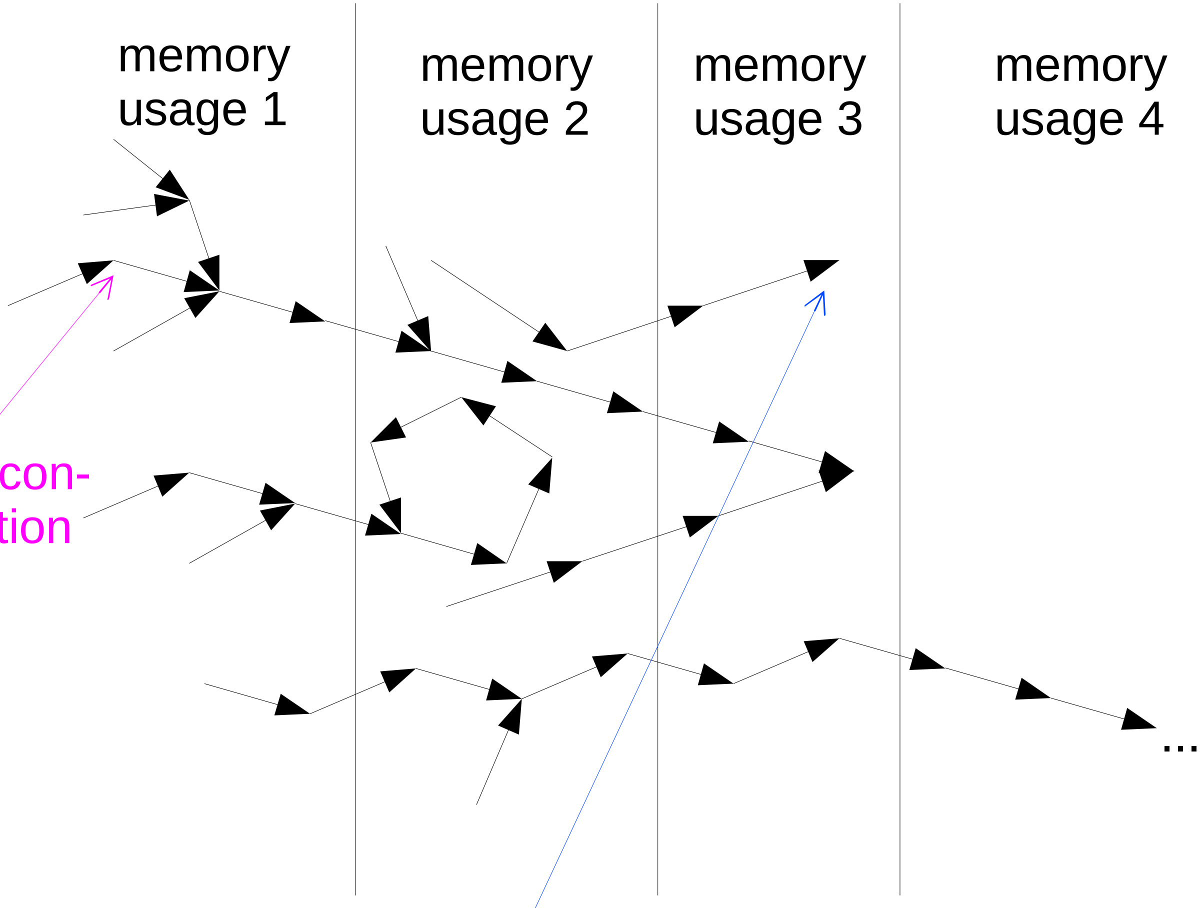
Sipser's theorem (✶)



k=3
Search(this_configuration)
Initial configuration not found

Memory usage is 3!
Now we search from accepting configurations

Sipser's theorem (✶)

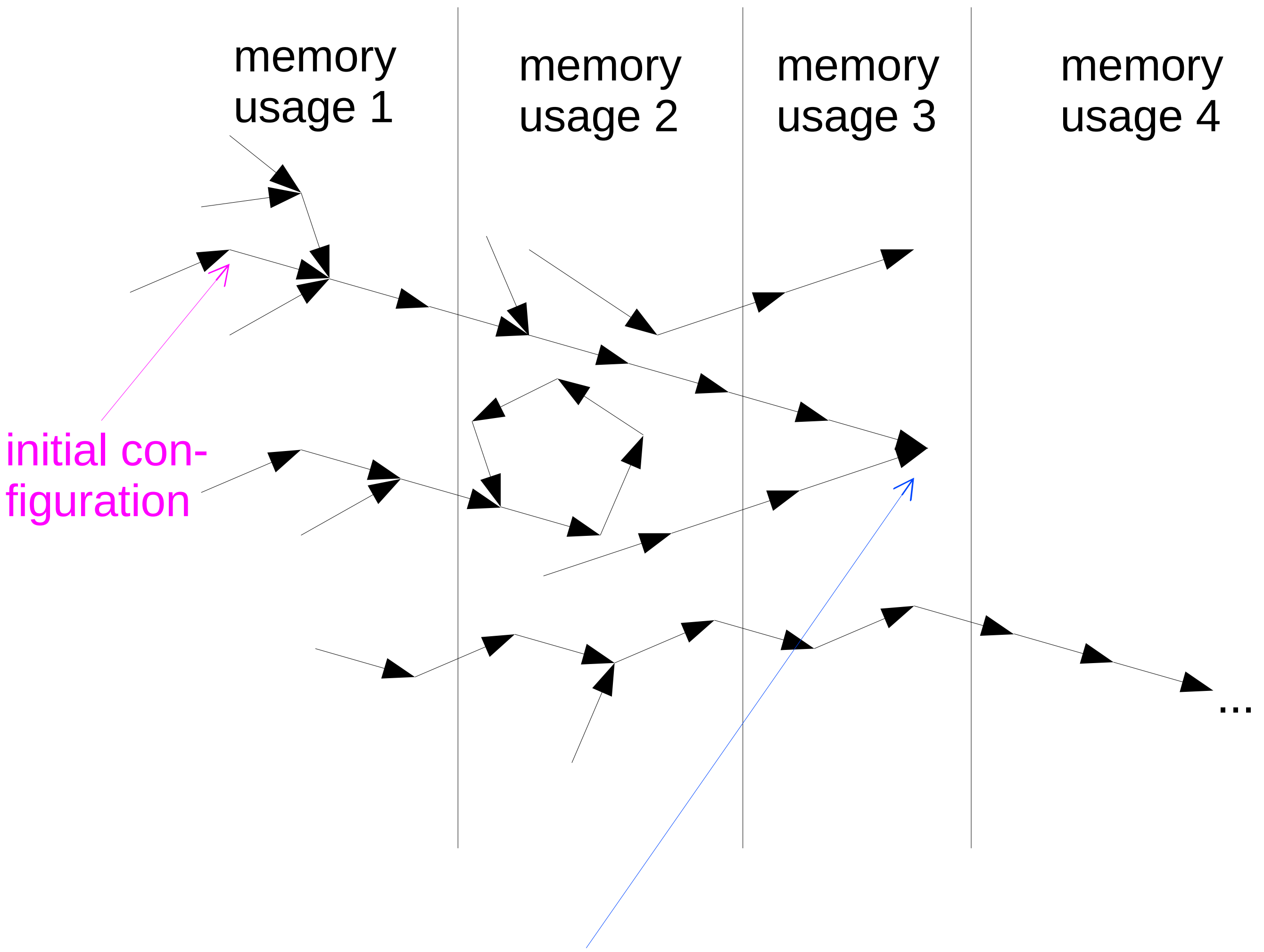


initial configuration

Memory usage is 3!
Now we search from
accepting configurations

Search(this_configuration)
Initial configuration not found

Sipser's theorem (✶)



initial configuration

Memory usage is 3!
Now we search from
accepting configurations

Search(this_configuration)
Initial configuration found!
There is an accepting run!

Corollary of the Sipser's theorem

If a language L is semidecidable, but not decidable, then every machine M recognizing L on some word w uses infinite memory.

Proof. If M uses only a finite memory on every input,
then M would work in space $S(n)$ for some function S.
By Sipser's theorem, L would be decidable.

Constructible functions

A function $f(n)$ is ***time-constructible*** if there exists a machine M, which for input 1^n

- outputs a word of length precisely $f(n)$,
- works in time $O(f(n))$.

Constructible functions

A function $f(n)$ is ***time-constructible*** if there exists a machine M, which for input 1^n

- outputs a word of length precisely $f(n)$,
- works in time $O(f(n))$.

Tutorials:

- If f and g are time-constructible, then $f+g$, $f \cdot g$, f^g as well
- Functions n, $\lfloor n \cdot log(n) \rfloor$, n^k, k^n are time-constructible

Function $\lfloor log(n) \rfloor$, nor any function asymptotically smaller than n, is not time-constructible.

Constructible functions

A function $f(n)$ is ***space-constructible*** if there exists a machine M, which for input 1^n

- outputs a word of length precisely $f(n)$,
- works in space $O(f(n))$.

Constructible functions

A function $f(n)$ is ***space-constructible*** if there exists a machine M, which for input 1^n

- outputs a word of length precisely $f(n)$,
- works in space $O(f(n))$.

Tutorials:

- Every time-constructible function is also space-constructible
- If f and g are space-constructible, then $f+g$, $f{\cdot}g$, f^g as well
- Functions n, $\lfloor log(n) \rfloor$, n^k, k^n are space-constructible

We will see soon that:

- The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible
- Neither are some very fast-growing functions

Constructible functions

A machine M works in space precisely $S(n)$.
Is then $S(n)$ space-constructible?

Constructible functions

A machine M works in space precisely $S(n)$.
Is then $S(n)$ space-constructible?

- At first glance – yes: for every n there is a word w of length n for which M uses precisely space $S(n)$.

Constructible functions

A machine M works in space precisely $S(n)$.
Is then $S(n)$ space-constructible?

- At first glance – yes: for every n there is a word w of length n for which M uses precisely space $S(n)$.
- But: while constructing the function $S(n)$ we get the word 1^n; not clear how to find the „worst" word w

Constructible functions

A machine M works in space precisely $S(n)$.
Is then $S(n)$ space-constructible?

- At first glance – yes: for every n there is a word w of length n for which M uses precisely space $S(n)$.
- But: while constructing the function $S(n)$ we get the word 1^n; not clear how to find the „worst” word w
- When $S(n)=\Omega(n)$, we can browse all words of length n and run M on each of them

Constructible functions

A machine M works in space precisely $S(n)$.
Is then $S(n)$ space-constructible?

- At first glance – yes: for every n there is a word w of length n for which M uses precisely space $S(n)$.
- But: while constructing the function $S(n)$ we get the word 1^n; not clear how to find the „worst" word w
- When $S(n)=\Omega(n)$, we can browse all words of length n and run M on each of them
- But if $S(n)$ is smaller, we cannot do this (next slides – an example)

Constructible functions

- Tutorials: There is a language, which is not regular, and which can be recognized in space $log(log(n))$. The machine recognizing it works in space (precisely) $\Theta(log(log(n)))$
- The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible

Constructible functions (✶)

The function $\lfloor log(log(n+2))\rfloor$ is not space-constructible.

<u>Proof.</u>

- To the contrary, suppose that M “constructs” $\lfloor log(log(n+2))\rfloor$
- M works in space $O(log(log(n+2)))$, so for large n it uses at most $c{\cdot}log(log(n))$ cells on inputs of length n, including cells on the output tape (for some constant c)

Constructible functions (⋆)

The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible.

Proof.

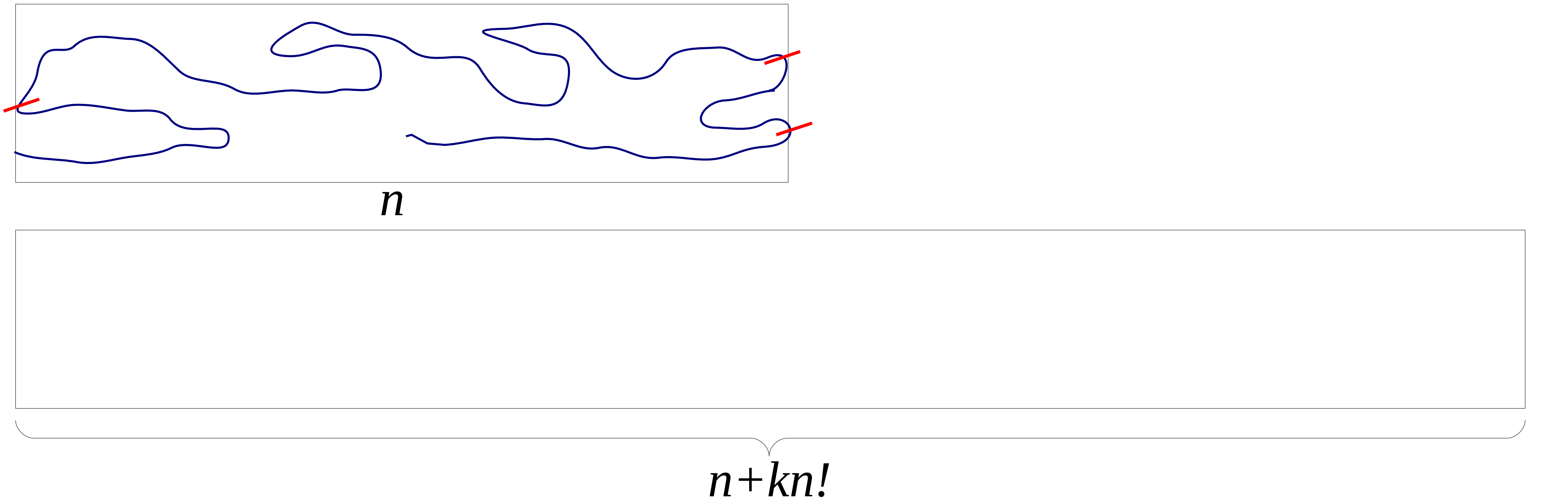
- To the contrary, suppose that M “constructs” $\lfloor log(log(n+2)) \rfloor$
- M works in space $O(log(log(n+2)))$, so for large n it uses at most $c \cdot log(log(n))$ cells on inputs of length n, including cells on the output tape (for some constant c)
- The number of all „internal” configurations (i.e., not counting the position of head on the input tape, but including the contents of the output tape) on inputs of length n is $(log(n))^d$ (for some constant d)
- Take $n>(log(n))^d$. We will prove that M produces the same output on $1^{n+kn!}$ for every k – contrary to the assumption

Constructible functions (⋆)

The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible.

Proof. Let $n>($number_of_internal_configurations_for_inputs_of_length_$n)$.
Consider the run of M on input 1^n.
We want to produce a run on input $1^{n+kn!}$, producing the same output.

- Cut the run on 1^n into fragments – split on configurations when we are over the first or over the last position of the input tape.

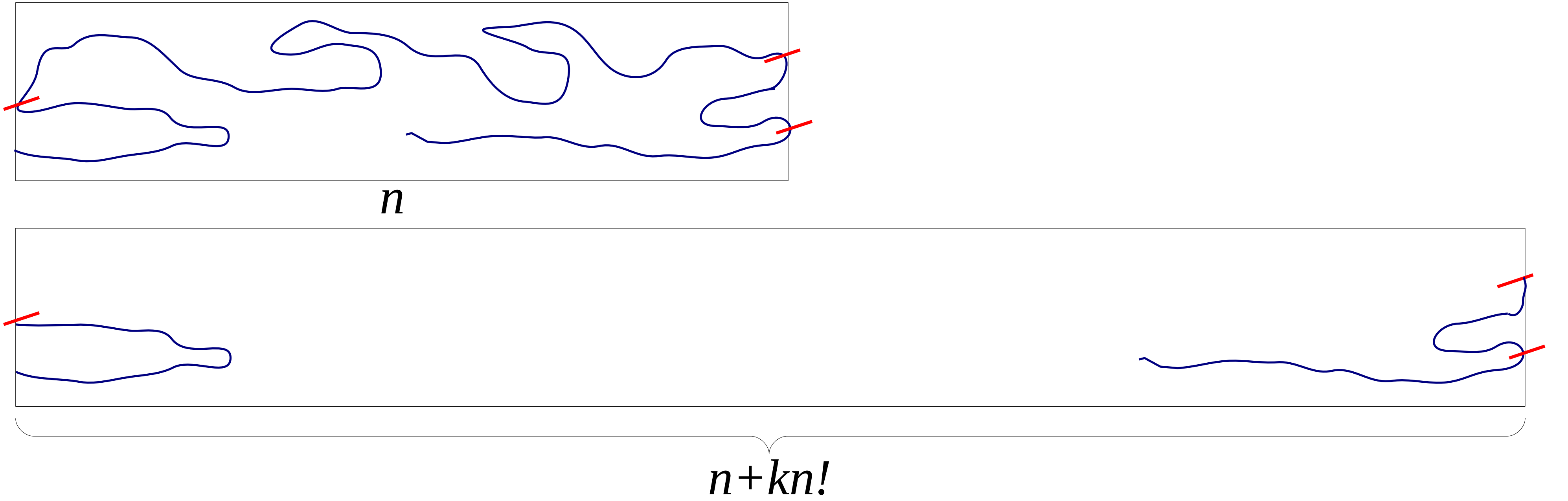


Constructible functions (⋆)

The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible.

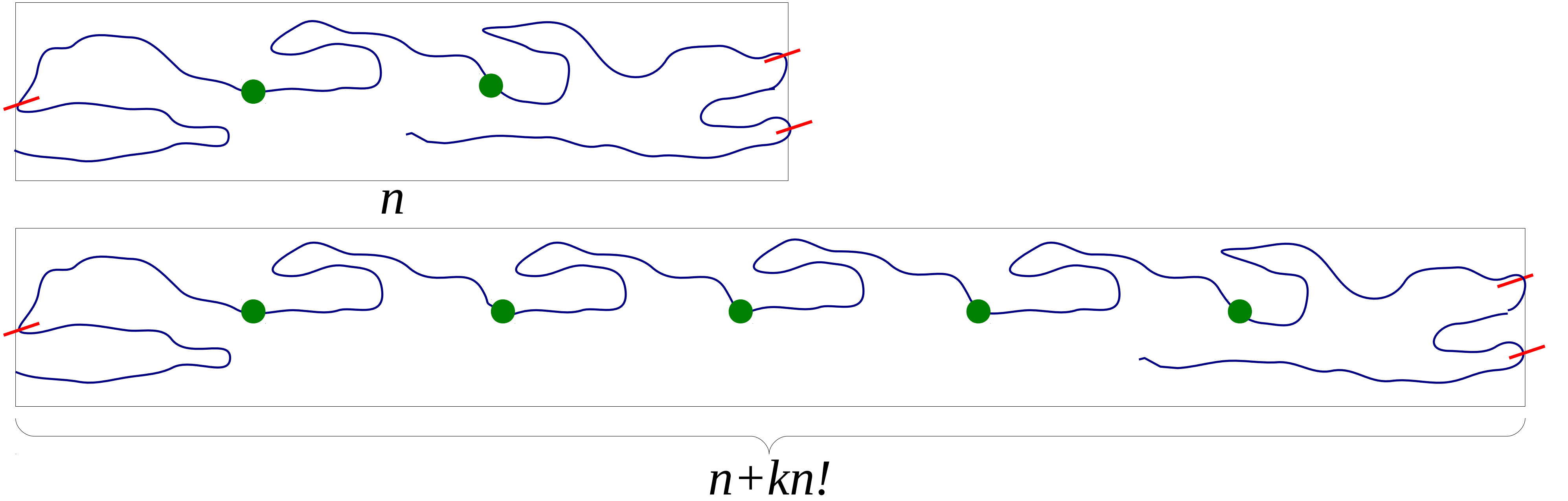
Proof. Let $n>($number_of_internal_configurations_for_inputs_of_length_$n)$.
Consider the run of M on input 1^n.
We want to produce a run on input $1^{n+kn!}$, producing the same output.

- Cut the run on 1^n into fragments – split on configurations when we are over the first or over the last position of the input tape.
- Fragments beginning and ending over the first position can be repeated when the input is $1^{n+kn!}$.
- Similarly fragments beginning and ending on the last position, and the last fragment



Constructible functions (⋆)

- Consider a fragment going from the beginning to the end (or vice versa)
- By the pigeonhole principle, there are two positions on the input tape such that M visits these positions in the same internal configuration.
- The part of the run between these two positions can be "pumped" (recall that the input word is uniform – contains only ones).
  The distance between these positions $m \leq n$ is a divisor of $(n+kn!)-n=kn!$
- (If there are multiple fragments crossing the whole word, they can be pumped in different places, no problem)
- Thus we have a run on input $1^{n+kn!}$, producing the same output as on 1^n (contradiction)



Constructible functions

- Tutorials: There is a language, which is not regular, and which can be recognized in space $log(log(n))$. The machine recognizing it works in space (precisely) $\Theta(log(log(n)))$
- The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible
- The same proof works for every unbounded nondecreasing function in $o(log(n))$

Constructible functions

- Tutorials: There is a language, which is not regular, and which can be recognized in space $log(log(n))$. The machine recognizing it works in space (precisely) $\Theta(log(log(n)))$
- The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible
- The same proof works for every unbounded nondecreasing function in $o(log(n))$
- But: there exists an unbouned function in $O(log(log(n)))$ which is space constructible (it is not nondecreasing)

Constructible functions

- Tutorials: There is a language, which is not regular, and which can be recognized in space $log(log(n))$. The machine recognizing it works in space (precisely) $\Theta(log(log(n)))$
- The function $\lfloor log(log(n+2)) \rfloor$ is not space-constructible
- The same proof works for every unbounded <u>nondecreasing</u> function in $o(log(n))$
- But: there exists an unbouned function in $O(log(log(n)))$ which is space constructible (it is not nondecreasing)
  This is:

$$S(n) = \lceil log(min\{i \mid i \text{ does not divide } n\}) \rceil$$