



Computational Complexity Theory

Lecture 2: Class NP, NP-completeness Cook-Levin theorem

Department of Computer Science,
Indian Institute of Science

Complexity class NP

Complexity Class NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.
- Class **NP** captures the set of decision problems whose solutions are *efficiently verifiable*.

Complexity Class NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.
- Class **NP** captures the set of decision problems whose solutions are *efficiently verifiable*.

Nondeterministic polynomial-time

Complexity Class NP

- **Definition.** A language $L \subseteq \{0,1\}^*$ is in **NP** if there's a polynomial function $p: \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial-time TM **M** (called the verifier) such that for every x ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

Complexity Class NP

- **Definition.** A language $L \subseteq \{0,1\}^*$ is in **NP** if there's a polynomial function $p: \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial-time TM **M** (called the verifier) such that for every x ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

u is called a certificate or witness for x (w.r.t L and M), if $x \in L$.

Complexity Class NP

- **Definition.** A language $L \subseteq \{0,1\}^*$ is in **NP** if there's a polynomial function $p: \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial-time TM **M** (called the verifier) such that for every x ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- It follows that verifier **M** cannot be fooled !

Complexity Class NP

- **Definition.** A language $L \subseteq \{0,1\}^*$ is in **NP** if there's a polynomial function $p: \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial-time TM **M** (called the verifier) such that for every x ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- Class **NP** contains those problems (languages) which have such efficient verifiers.

Class NP : Examples

- Vertex cover
 - Given a graph G and an integer k , check if G has a vertex cover of size k .

Class NP : Examples

- Vertex cover
- 0/1 integer programming
 - Given a system of linear (in)equalities with integer coefficients, check if there's a **0-1** assignment to the variables that satisfy all the (in)equalities.

Class NP : Examples

- Vertex cover
- 0/1 integer programming
- Integer factoring
 - Given two numbers n and U , check if n has a prime factor less than or equal to U .

Class NP : Examples

- Vertex cover
- 0/1 integer programming
- Integer factoring
- Graph isomorphism
 - Given two graphs, check if they are isomorphic.

Class NP : Examples

- 2-Diophantine solvability
 - Given three integers a , b and c , check if the equation $ax^2 + by + c = 0$ has a solution (x, y) , where both x and y are positive integers.

[Homework]: Show that the above problem is in NP.

Is $P = NP$?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!

Is $P = NP$?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- Solving a problem does seem harder than verifying its solution, so most people believe that $P \neq NP$.

Is $P = NP$?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- $P = NP$ has many weird consequences, and if true, will pose a serious threat to secure and efficient cryptography (and e-commerce).

Is $P = NP$?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- Mathematics has gained much from attempts to prove such “negative” statements—Galois theory, Godel’s incompleteness, Fermat’s Last Theorem, Turing’s undecidability, Continuum hypothesis etc.

Is $P = NP$?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- Complexity theory has several of such intriguing unsolved questions.

The history and status of the P versus NP question

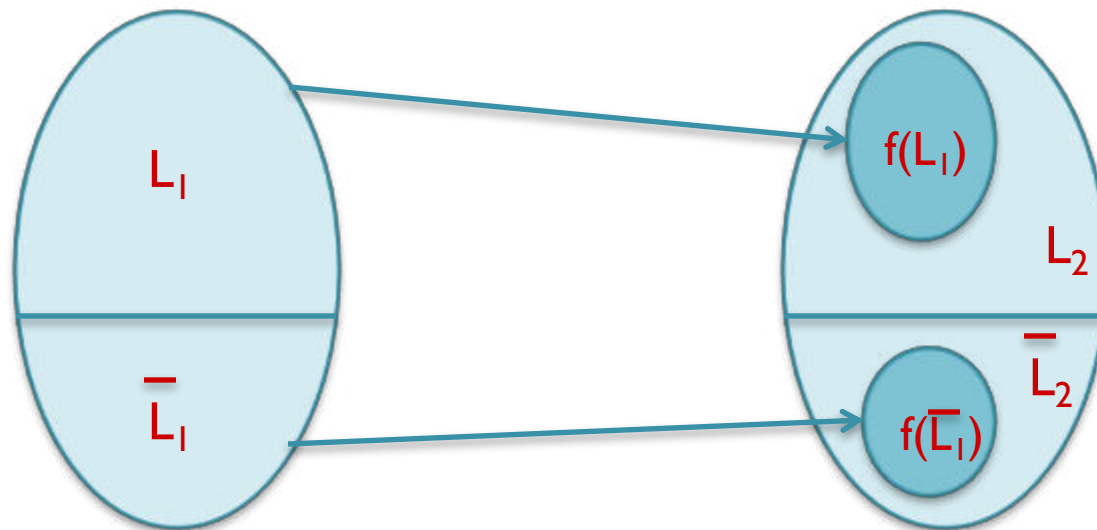
– survey by Michael Sipser (1992)

Reductions

Polynomial-time reduction

- **Definition.** We say a language $L_1 \subseteq \{0,1\}^*$ is polynomial-time (Karp) reducible to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial-time computable function f s.t.

$$x \in L_1 \iff f(x) \in L_2$$



Polynomial-time reduction

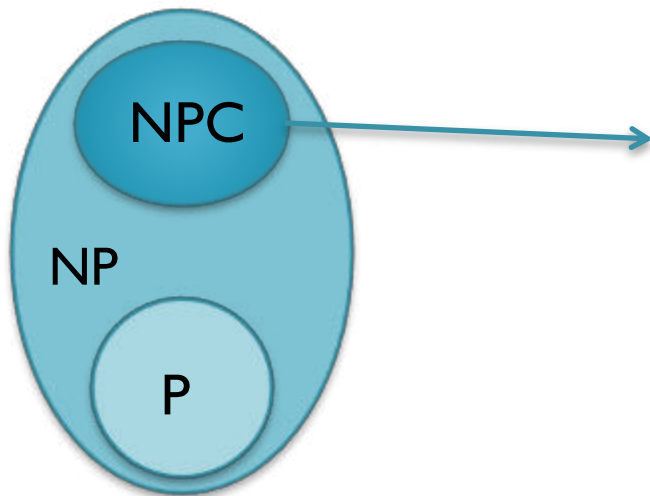
- **Definition.** We say a language $L_1 \subseteq \{0,1\}^*$ is polynomial-time (Karp) reducible to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.

$$x \in L_1 \iff f(x) \in L_2$$

- **Notation.** $L_1 \leq_p L_2$
- **Observe.** If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.

NP-completeness

- **Definition.** A language L' is *NP-hard* if for every L in NP , $L \leq_p L'$. Further, L' is *NP-complete* if L' is in NP and is NP-hard.
- **Observe.** If L' is NP-hard and L' is in P then $P = NP$. If L' is NP-complete then L' is in P if and only if $P = NP$.



Hardest problems inside NP in the sense that if one NPC problem is in P then all problems in NP is in P .

NP-completeness

- **Definition.** A language L' is *NP-hard* if for every L in NP , $L \leq_p L'$. Further, L' is *NP-complete* if L' is in NP and is NP-hard.
- **Observe.** If L' is NP-hard and L' is in P then $P = NP$. If L' is NP-complete then L' in P if and only if $P = NP$.
- **[Homework].** Let $L_1 \subseteq \{0,1\}^*$ be any language and L_2 be a language in NP . If $L_1 \leq_p L_2$ then L_1 is also in NP .

Few words on reductions

- As to how we define a reduction from one language to the other (or one function to the other) is usually guided by a question on whether two complexity classes are different or identical.
- For polynomial-time reductions, the question is whether or not P equals NP .
- Reductions help us define *complete problems* (the 'hardest' problems in a class) which in turn help us compare the complexity classes under consideration.

Class NP : Examples

- Vertex cover (NP-complete)
- 0/1 integer programming (NP-complete)
- 3-coloring planar graphs (NP-complete)
- 2-Diophantine solvability (NP-complete)
- Integer factoring (unlikely to be NP-complete)
- Graph isomorphism (Quasi-P) *Babai 2015*

How to show existence of an NPC problem?

- Let $L' = \{ (\alpha, x, l^m, l^t) : \text{there exists a } u \in \{0,1\}^m \text{ s.t. } M_\alpha \text{ accepts } (x, u) \text{ in } t \text{ steps} \}$
- **Observation.** L' is NP-complete.
- The language L' involves Turing machine in its definition. Next, we'll see an example of an NP-complete problem that is arguably more natural.

A natural NP-complete problem

- **Definition.** A Boolean formula on variables x_1, \dots, x_n consists of AND, OR and NOT operations.

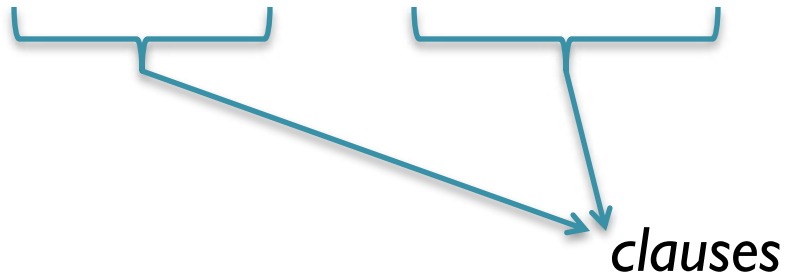
e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- **Definition.** A Boolean formula ϕ is satisfiable if there's a $\{0,1\}$ -assignment to its variables that makes ϕ evaluate to 1.

A natural NP-complete problem

- **Definition.** A Boolean formula is in Conjunctive Normal Form (CNF) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$



A natural NP-complete problem

- **Definition.** A Boolean formula is in Conjunctive Normal Form (CNF) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$



literals

A natural NP-complete problem

- **Definition.** A Boolean formula is in Conjunctive Normal Form (CNF) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- **Definition.** Let **SAT** be the language consisting of all *satisfiable CNF formulae*.

A natural NP-complete problem

- **Definition.** A Boolean formula is in Conjunctive Normal Form (CNF) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- **Definition.** Let **SAT** be the language consisting of all *satisfiable CNF formulae*.
- **Theorem.** (Cook 1971, Levin 1973) **SAT** is NP-complete.

A natural NP-complete problem

- **Definition.** A Boolean formula is in Conjunctive Normal Form (CNF) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- **Definition.** Let **SAT** be the language consisting of all *satisfiable CNF formulae*.

- **Theorem.** (Cook 1971, Levin 1973) **SAT** is NP-complete.

Easy to see that **SAT** is in **NP**.

Need to show that **SAT** is NP-hard.

Proof of Cook-Levin Theorem

Cook-Levin theorem: Proof

- **Main idea:** Computation is *local*; i.e., every step of computation *looks at* and *changes* only constantly many bits; and this step can be implemented by a small CNF formula.

Cook-Levin theorem: Proof

- **Main idea:** Computation is **local**; i.e., every step of computation *looks at* and *changes* only constantly many bits; and this step can be implemented by a small CNF formula.
- Let $L \in NP$. We intend to come up with a polynomial-time computable function $f: x \mapsto \phi_x$ s.t.,
 - $x \in L \iff \phi_x \in SAT$

Cook-Levin theorem: Proof

- **Main idea:** Computation is **local**; i.e., every step of computation *looks at* and *changes* only constantly many bits; and this step can be implemented by a small CNF formula.
- Let $L \in NP$. We intend to come up with a polynomial-time computable function $f: x \mapsto \phi_x$ s.t.,
 - $x \in L \iff \phi_x \in SAT$
 - Notation: $|\phi_x| :=$ size of ϕ_x
= number of \vee or \wedge in ϕ_x

Cook-Levin theorem: Proof

- Language L has a poly-time verifier M such that
$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

Cook-Levin theorem: Proof

- Language L has a poly-time verifier M such that

$$x \in L \iff \exists u \in \{0, 1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- **Idea:** For any fixed x , capture the computation of $M(x, \cdot)$ by a CNF ϕ_x such that

$$\exists u \in \{0, 1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1 \iff \phi_x \text{ is satisfiable}$$

Cook-Levin theorem: Proof

- Language L has a poly-time verifier M such that

$$x \in L \iff \exists u \in \{0, 1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- Idea:** For any fixed x , capture the computation of $M(x, ..)$ by a CNF ϕ_x such that

$$\exists u \in \{0, 1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1 \iff \phi_x \text{ is satisfiable}$$

- For any fixed x , $M(x, ..)$ is a deterministic TM that takes u as input and runs in time polynomial in $|u|$.

Cook-Levin theorem: Proof

- **Main Theorem.** Let N be a deterministic TM that runs in time $T(n)$ on every input u of length n , and outputs $0/1$. Then,

Cook-Levin theorem: Proof

- **Main Theorem.** Let N be a deterministic TM that runs in time $T(n)$ on every input u of length n , and outputs 0/1. Then,
 1. There's a CNF $\phi(u, \text{"auxiliary variables"})$ of size $\text{poly}(T(n))$ such that for any fixed u , $\phi(u, \text{"auxiliary variables"})$ is satisfiable as a function of the "auxiliary variables" if and only if $N(u) = 1$.
 2. ϕ is computable in time $\text{poly}(T(n))$ from N and n .

Cook-Levin theorem: Proof

- **Main Theorem.** Let N be a deterministic TM that runs in time $T(n)$ on every input u of length n , and outputs $0/1$. Then,
 1. There's a CNF $\phi(u, \text{"auxiliary variables"})$ of size $\text{poly}(T(n))$ such that for any fixed u , $\phi(u, \text{"auxiliary variables"})$ is satisfiable as a function of the "auxiliary variables" if and only if $N(u) = 1$.
 2. ϕ is computable in time $\text{poly}(T(n))$ from N and n .
- $\phi(u, \text{"auxiliary variables"})$ is satisfiable as a function of **all** variables if and only if $\exists u$ s.t $N(u) = 1$.

Cook-Levin theorem: Proof

- **Main Theorem.** Let N be a deterministic TM that runs in time $T(n)$ on every input u of length n , and outputs 0/1. Then,
 1. There's a CNF $\phi(u, \text{"auxiliary variables"})$ of size $\text{poly}(T(n))$ such that for any fixed u , $\phi(u, \text{"auxiliary variables"})$ is satisfiable as a function of the "auxiliary variables" if and only if $N(u) = 1$.
 2. ϕ is computable in time $\text{poly}(T(n))$ from N and n .
- Cook-Levin theorem follows from above!

Proof of Main Theorem

Main theorem: Proof

- Step 1. Let N be a deterministic TM that runs in time $T(n)$ on every input u of length n , and outputs $0/1$. Then,
 1. There's a Boolean circuit ψ of size $\text{poly}(T(n))$ such that $\psi(u) = 1$ if and only if $N(u) = 1$.
 2. ψ is computable in time $\text{poly}(T(n))$ from N & n .
- Step 2. “Convert” circuit ψ to a CNF ϕ efficiently by introducing auxiliary variables.

Main theorem: Step I

- Assume (w.l.o.g) that **N** has a single tape and it writes its output on the first cell at the end of computation.

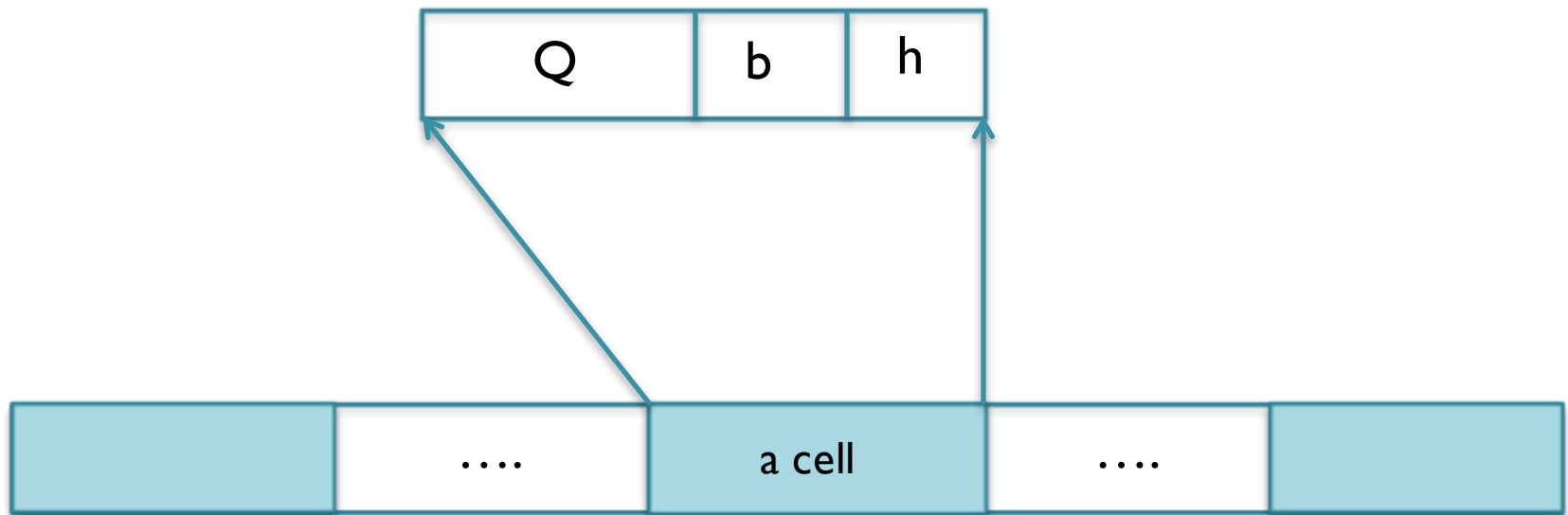
Main theorem: Step I

- Assume (w.l.o.g) that **N** has a single tape and it writes its output on the first cell at the end of computation.
- A step of computation of **N** consists of
 - Changing the content of the current cell
 - Changing state
 - Changing head position

Main theorem: Step I

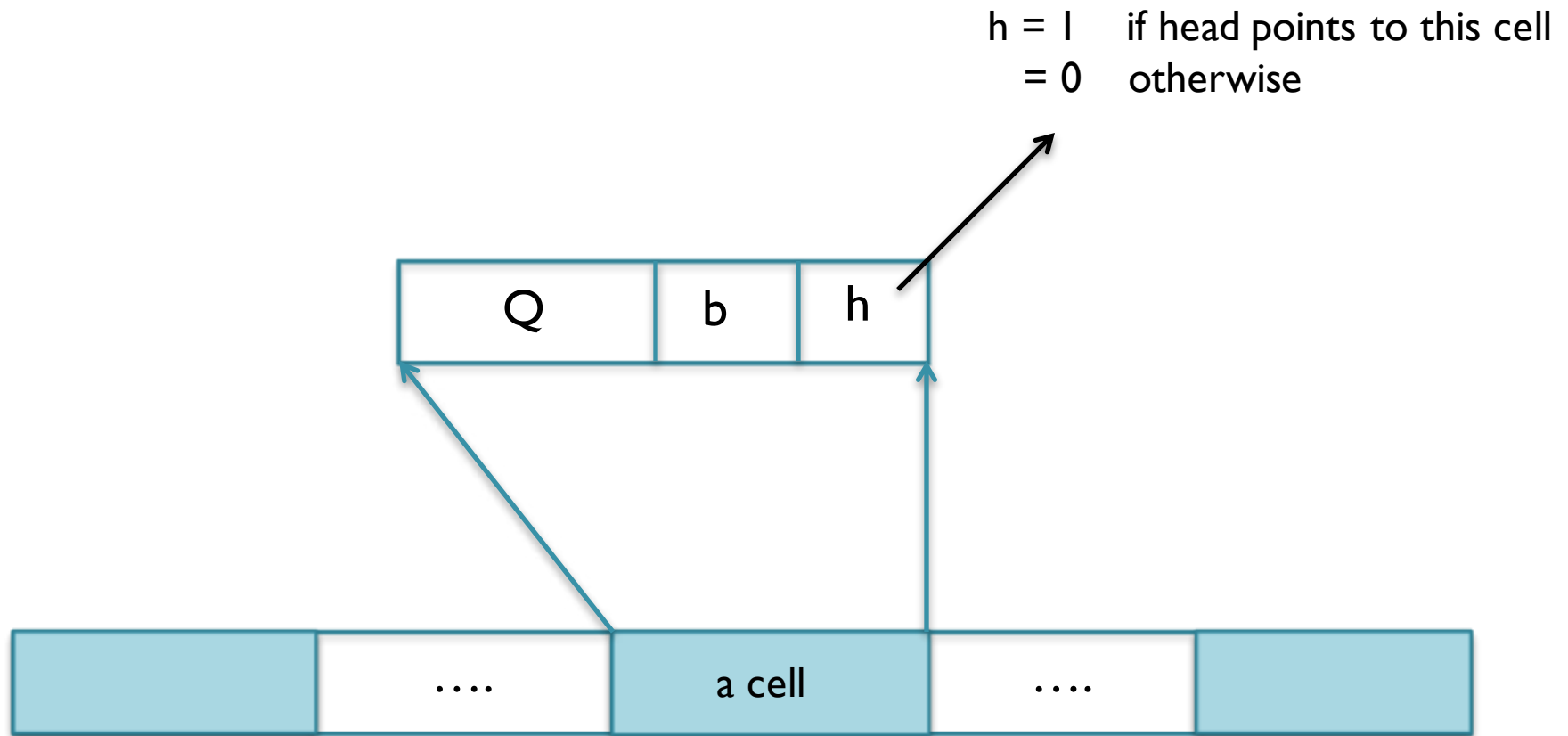
- Assume (w.l.o.g) that **N** has a single tape and it writes its output on the first cell at the end of computation.
- A step of computation of **N** consists of
 - Changing the content of the current cell
 - Changing state
 - Changing head position
- Think of a 'compound' tape: Every cell stores the current state, a bit content and head indicator.

Main theorem: Step I



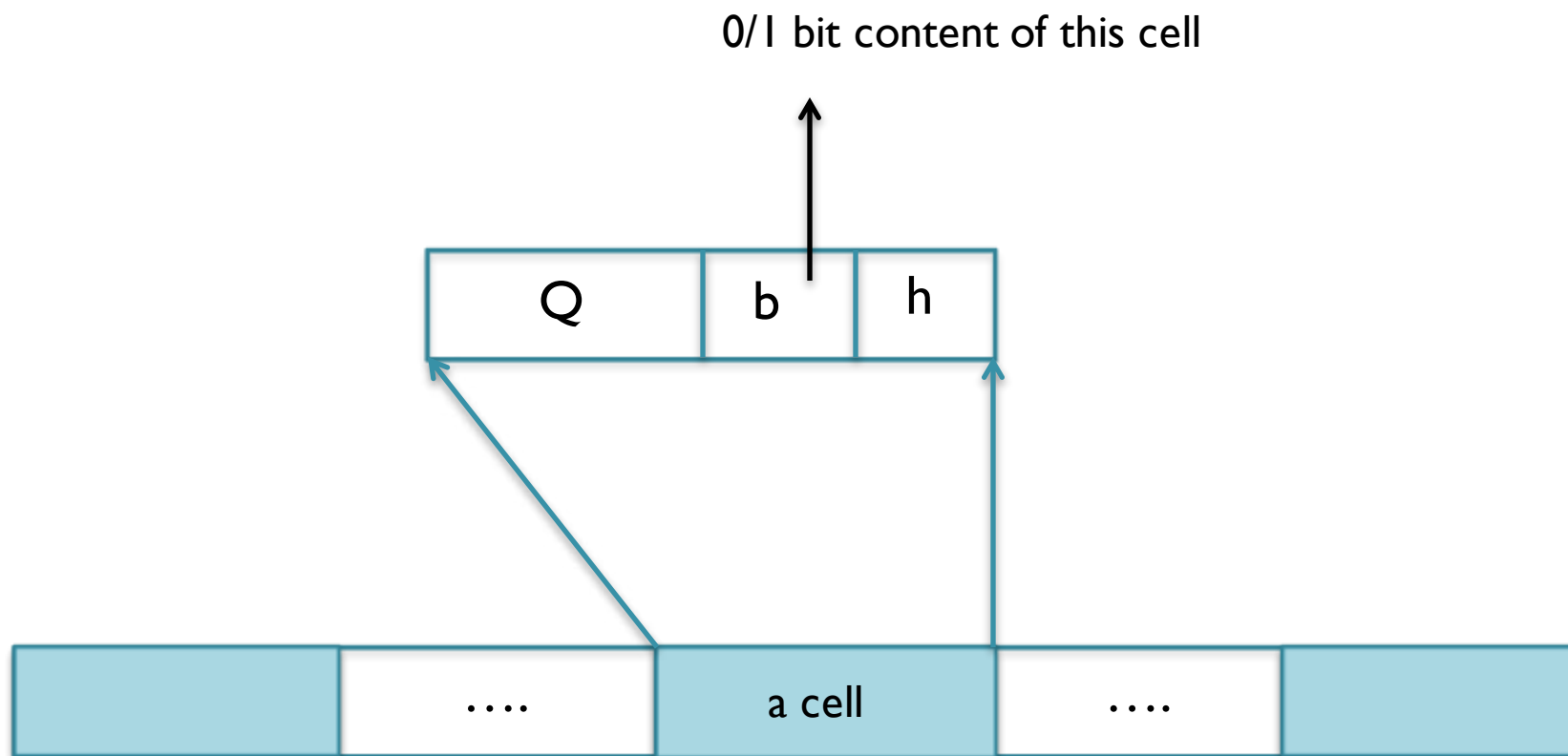
A compound tape

Main theorem: Step I



A compound tape

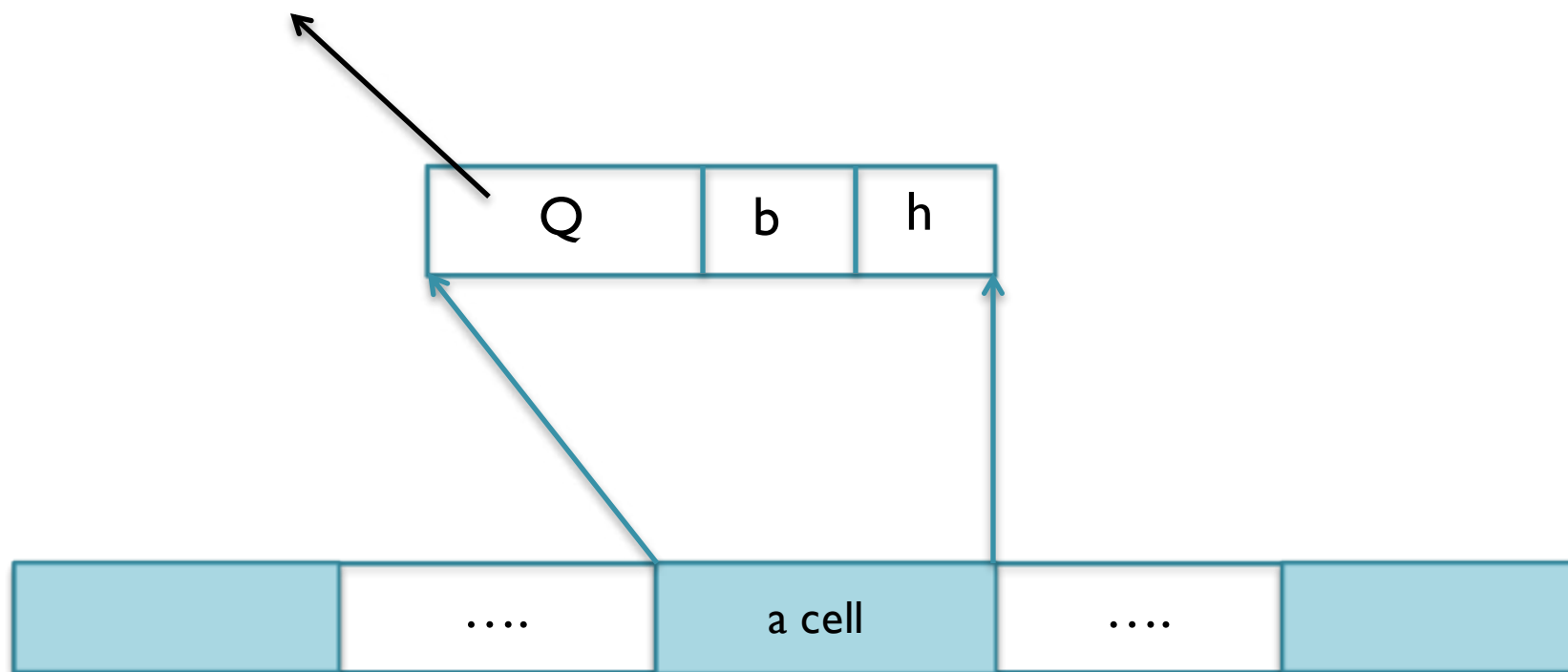
Main theorem: Step I



A compound tape

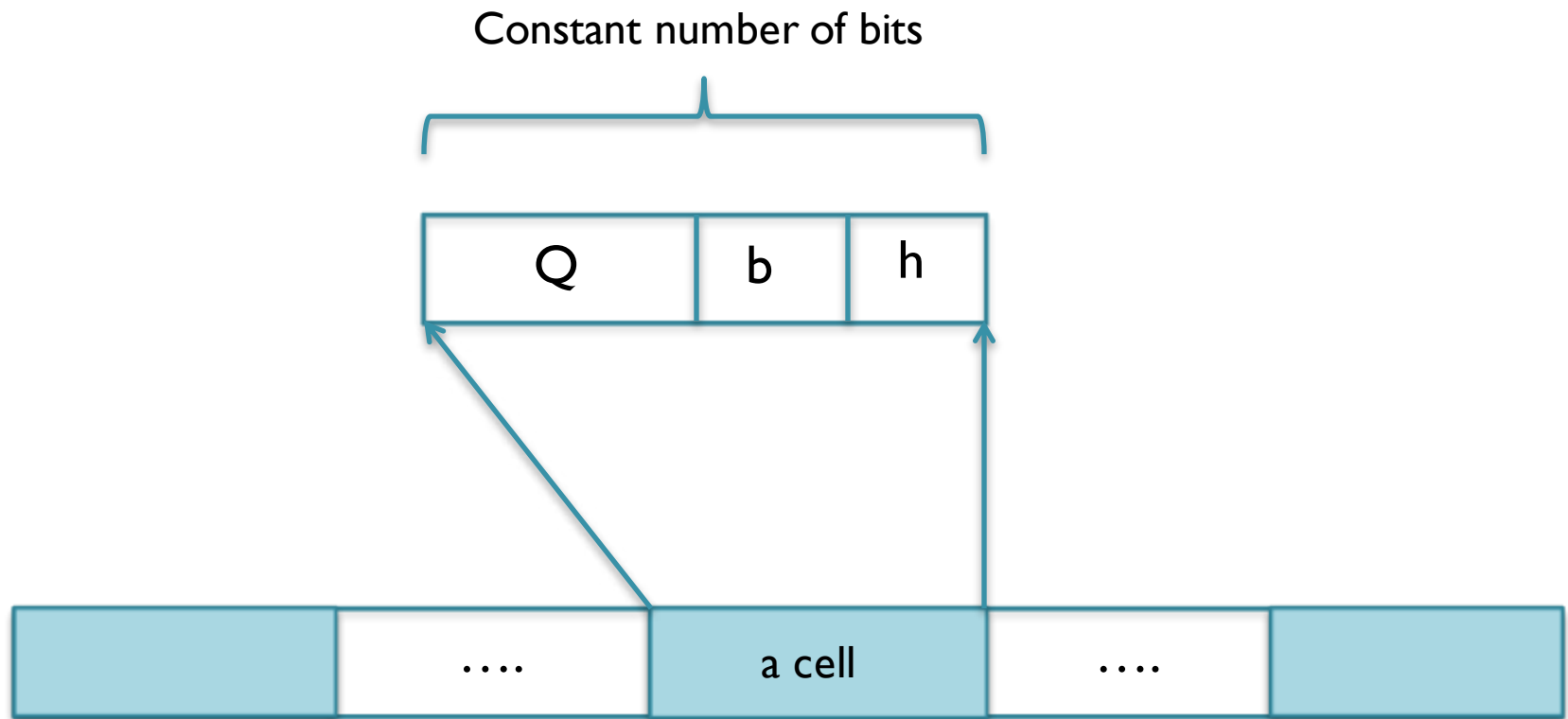
Main theorem: Step I

Current state when $h = 1$



A compound tape

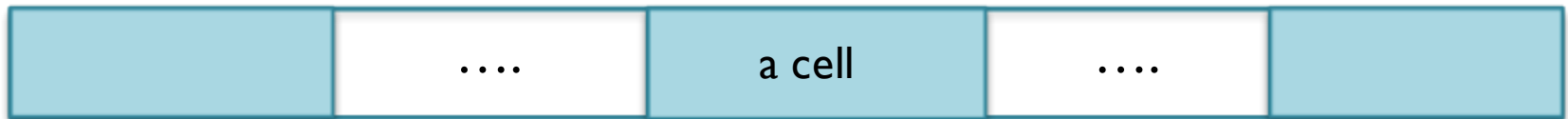
Main theorem: Step I



A compound tape

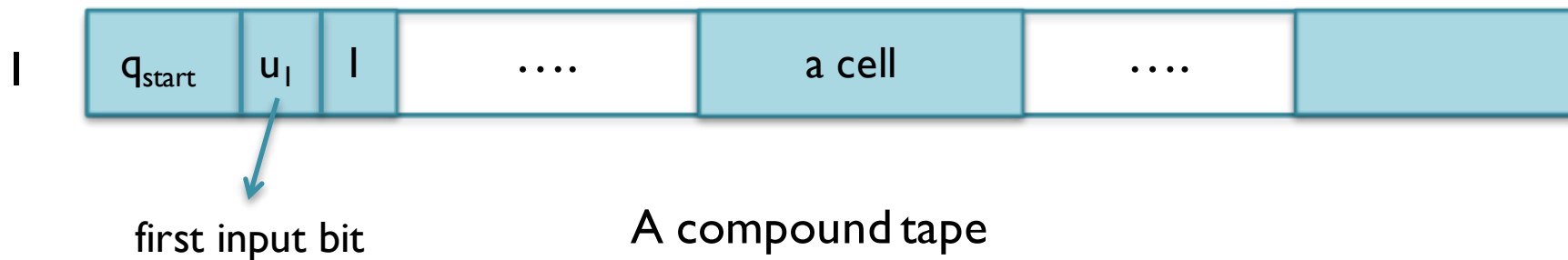
Main theorem: Step I

- Computation of N on inputs of length n can be completely described by a sequence of $T(n)$ compound tapes, the i -th of which captures a 'snapshot' of N 's computation at the i -th step.

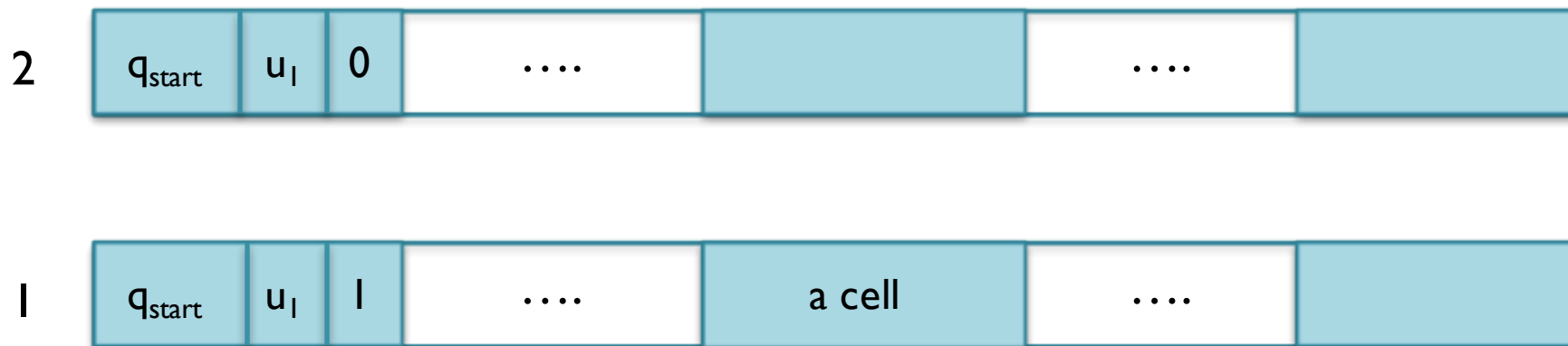


A compound tape

Main theorem: Step I

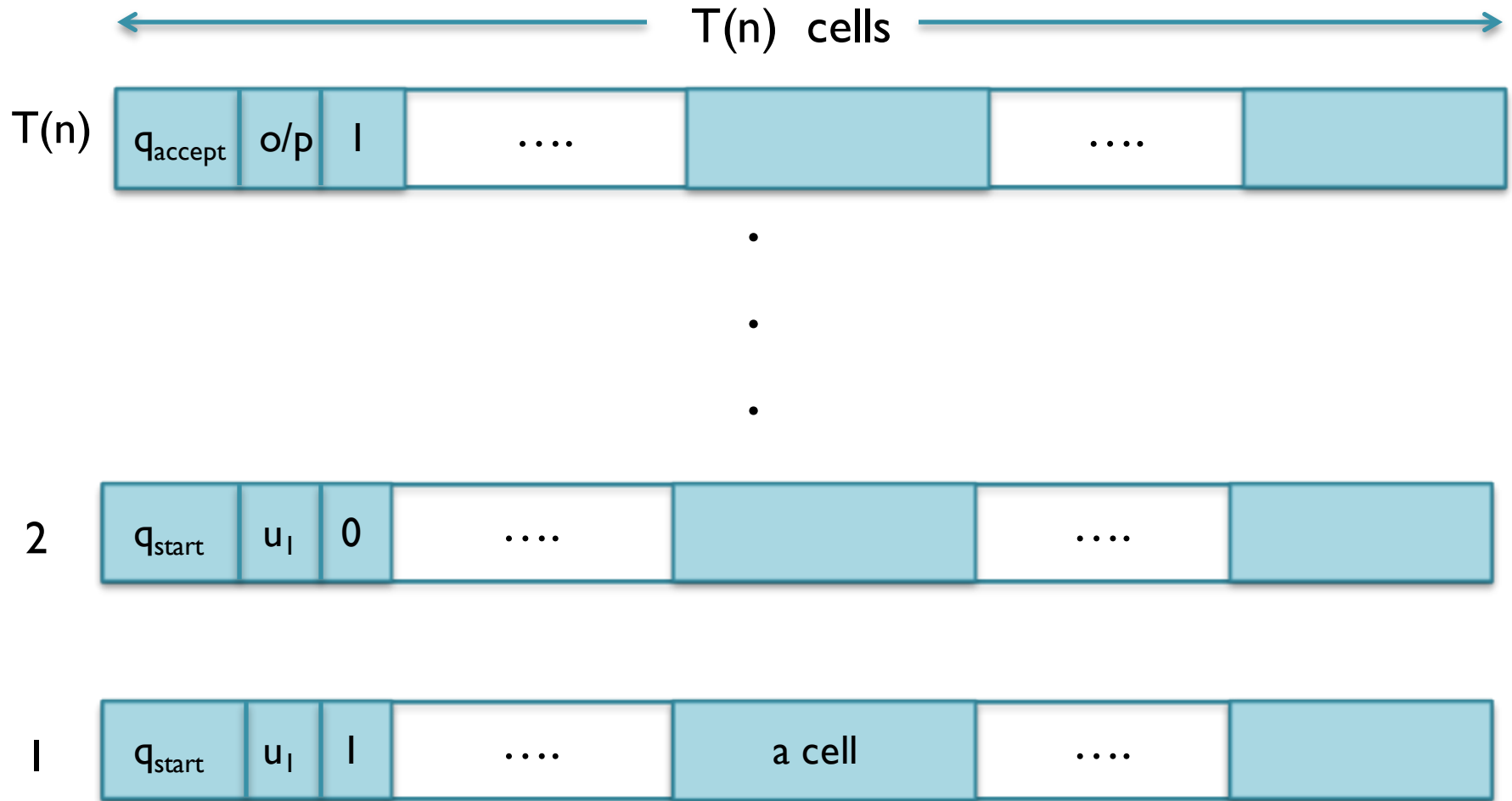


Main theorem: Step I



A compound tape

Main theorem: Step I

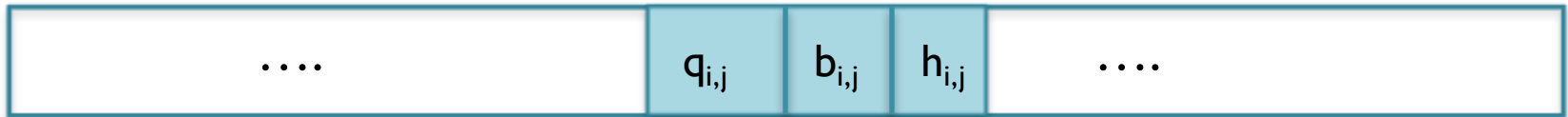


A compound tape

Main theorem: Step I

- $h_{i,j} = 1$ iff head points to cell j at i -th step
- $b_{i,j}$ = bit content of cell j at i -th step
- $q_{i,j}$ = a sequence of $\log |Q|$ bits which contains the current state info if $h_{i,j} = 1$; otherwise we don't care

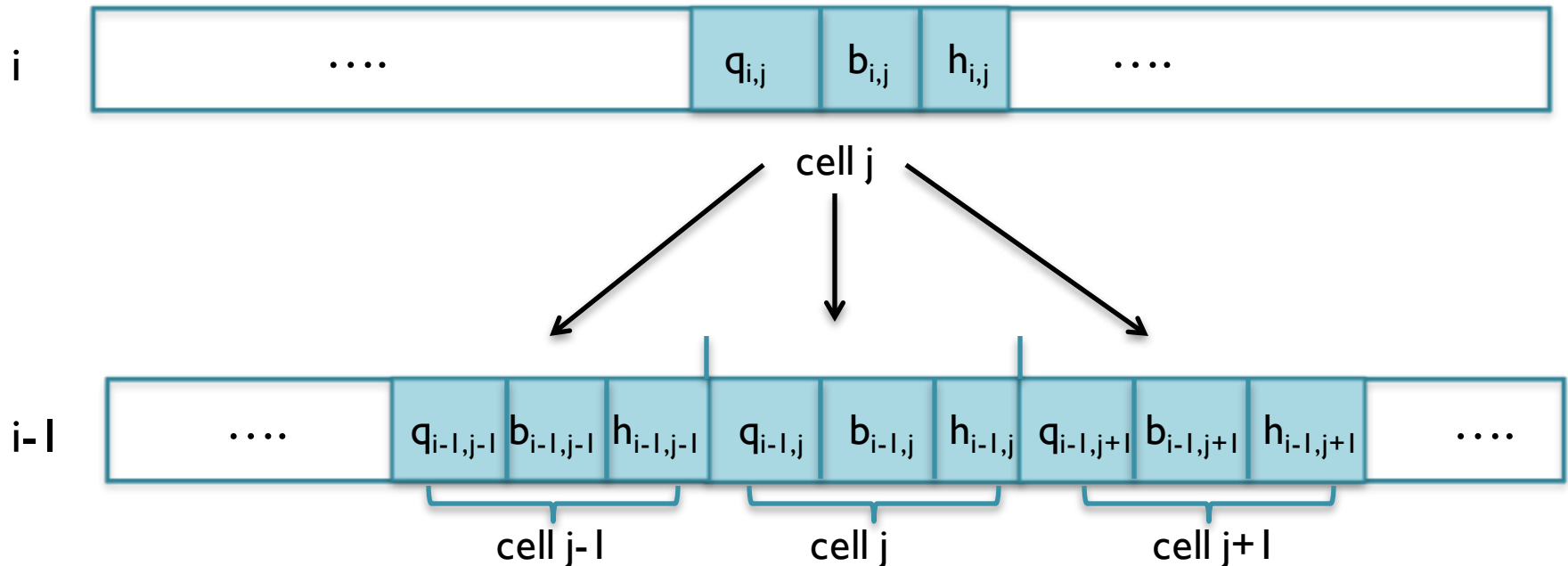
i



cell j

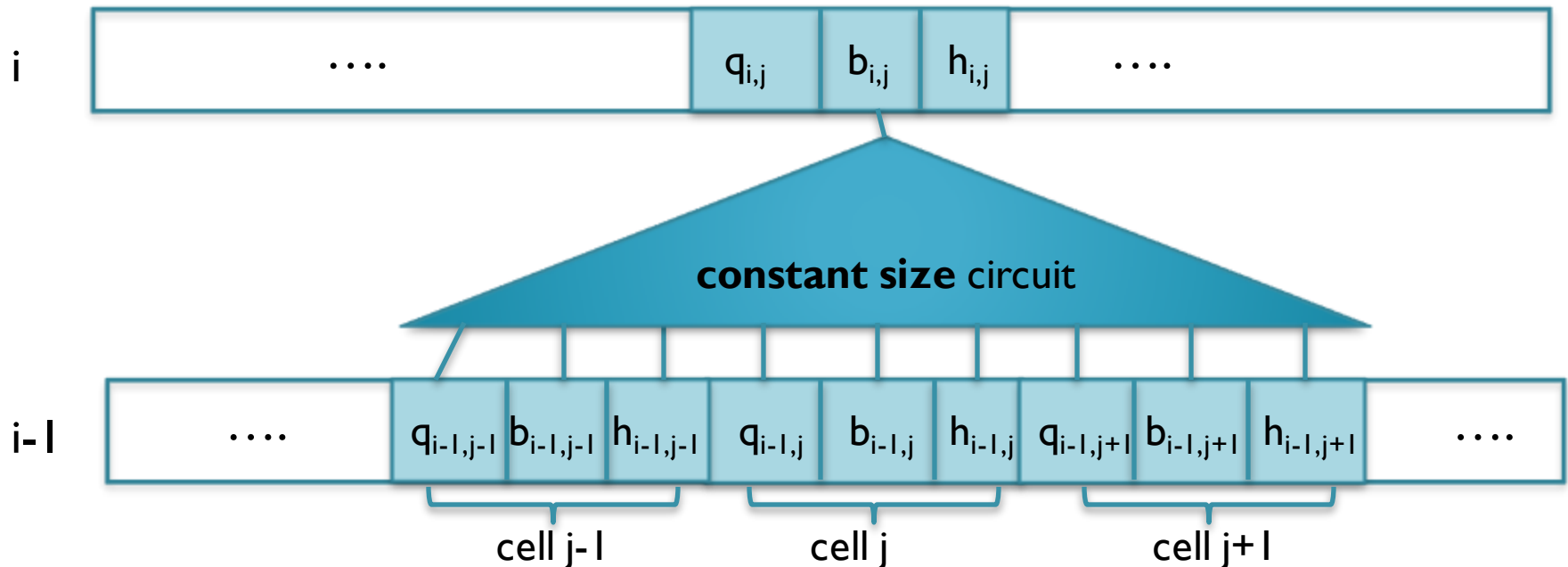
Main theorem: Step I

- **Locality of computation:** The bits in $h_{i,j}$, $b_{i,j}$ and $q_{i,j}$ depend only on the bits in
 - $h_{i-1,j-1}$, $b_{i-1,j-1}$, $q_{i-1,j-1}$,
 - $h_{i-1,j}$, $b_{i-1,j}$, $q_{i-1,j}$,
 - $h_{i-1,j+1}$, $b_{i-1,j+1}$, $q_{i-1,j+1}$

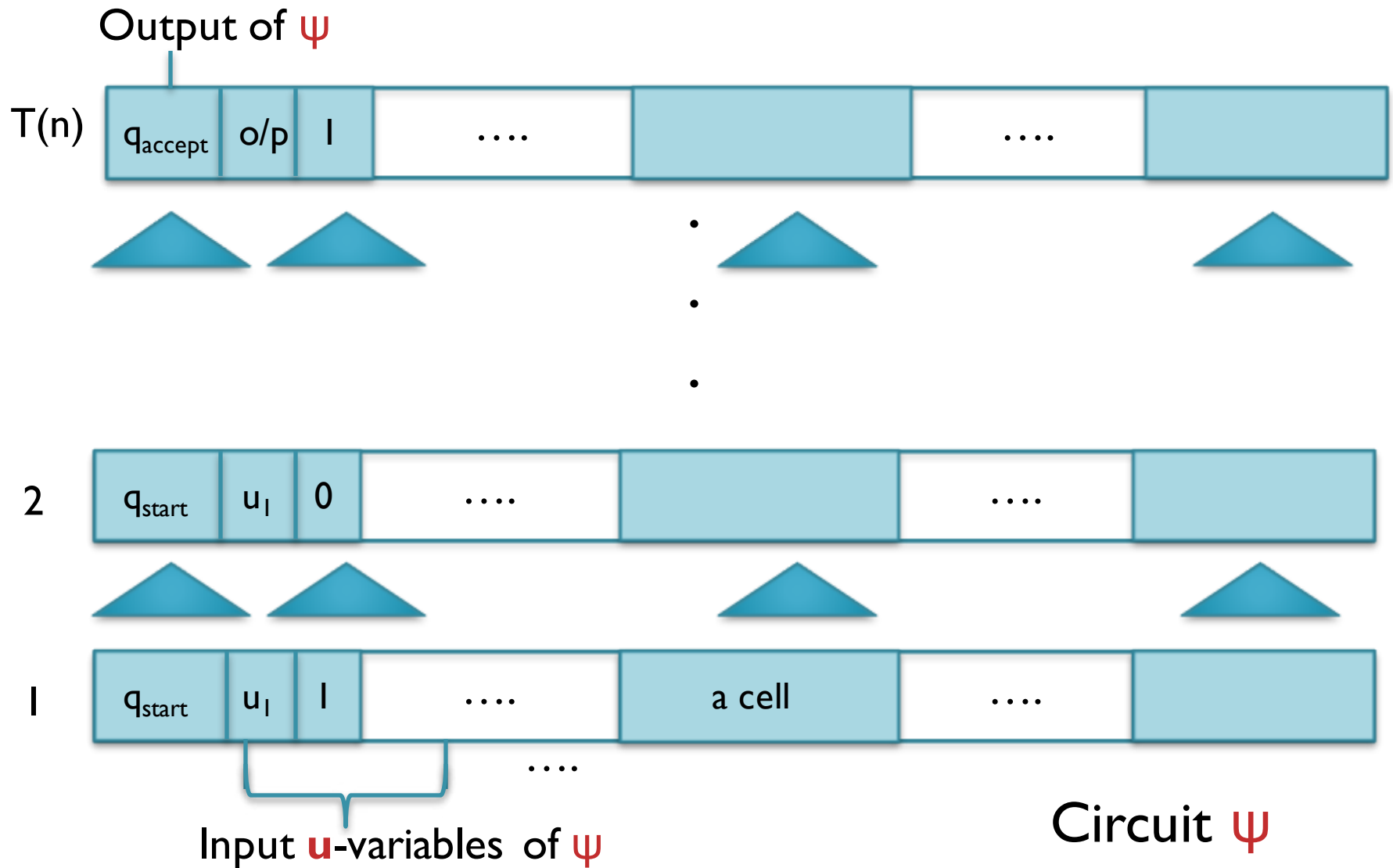


Main theorem: Step I

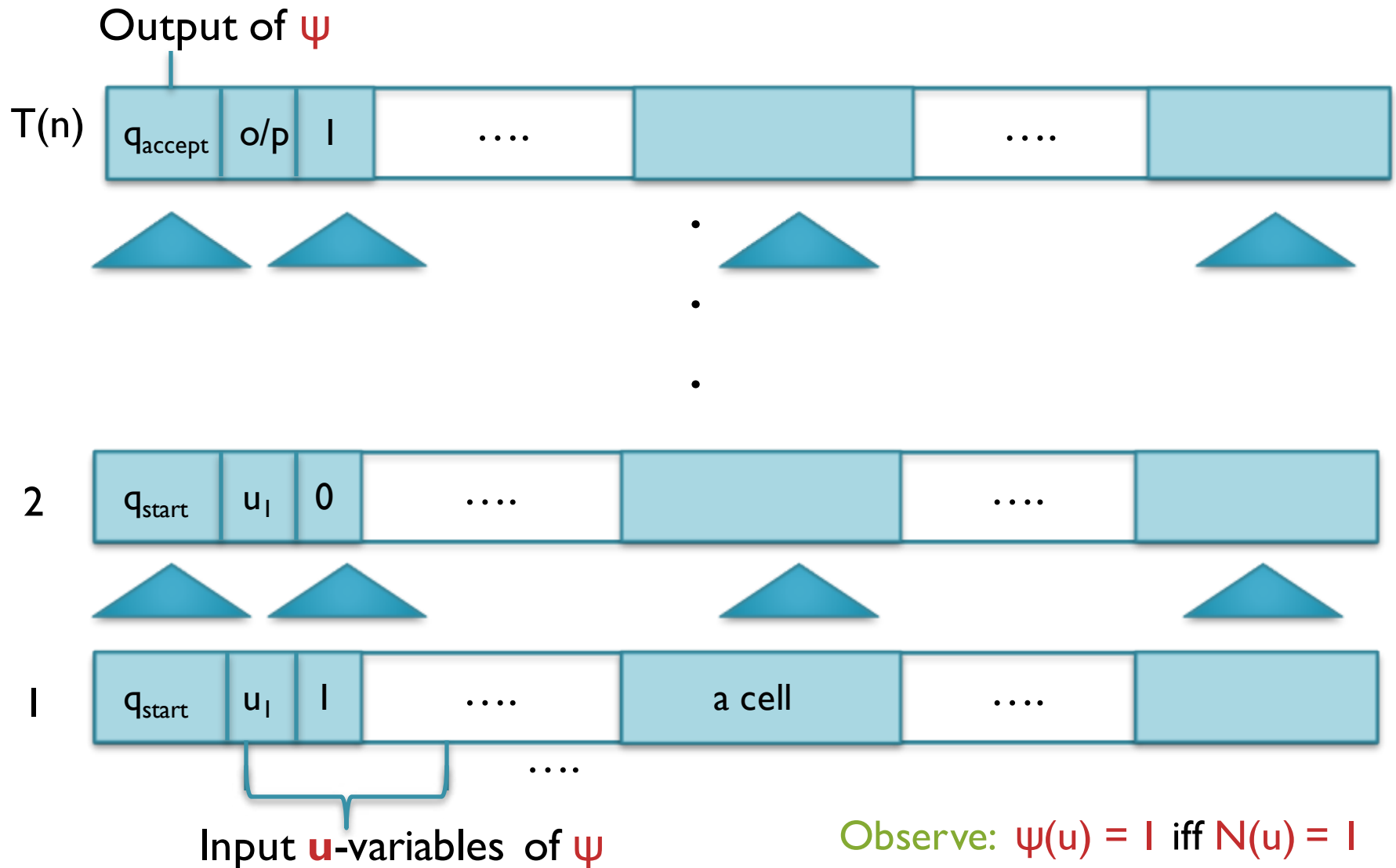
- **Locality of computation:** The bits in $h_{i,j}$, $b_{i,j}$ and $q_{i,j}$ depend only on the bits in
 - $h_{i-1,j-1}$, $b_{i-1,j-1}$, $q_{i-1,j-1}$,
 - $h_{i-1,j}$, $b_{i-1,j}$, $q_{i-1,j}$,
 - $h_{i-1,j+1}$, $b_{i-1,j+1}$, $q_{i-1,j+1}$



Main theorem: Step I



Main theorem: Step I



Recall Steps 1 and 2

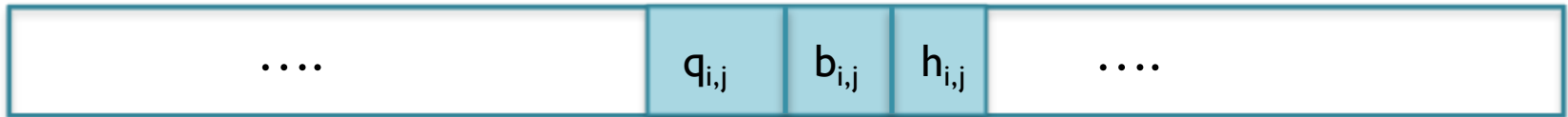
- **Step 1.** Let N be a deterministic TM that runs in time $T(n)$ on every input u of length n , and outputs $0/1$. Then,
 1. There's a Boolean circuit ψ of size $\text{poly}(T(n))$ such that $\psi(u) = 1$ if and only if $N(u) = 1$.
 2. ψ is computable in time $\text{poly}(T(n))$ from N & n .
- **Step 2.** “Convert” circuit ψ to a CNF ϕ efficiently by introducing auxiliary variables.

Main theorem: Step 2

- Think of $h_{i,j}$, $b_{i,j}$ and the bits of $q_{i,j}$ as formal Boolean variables.

auxiliary variables

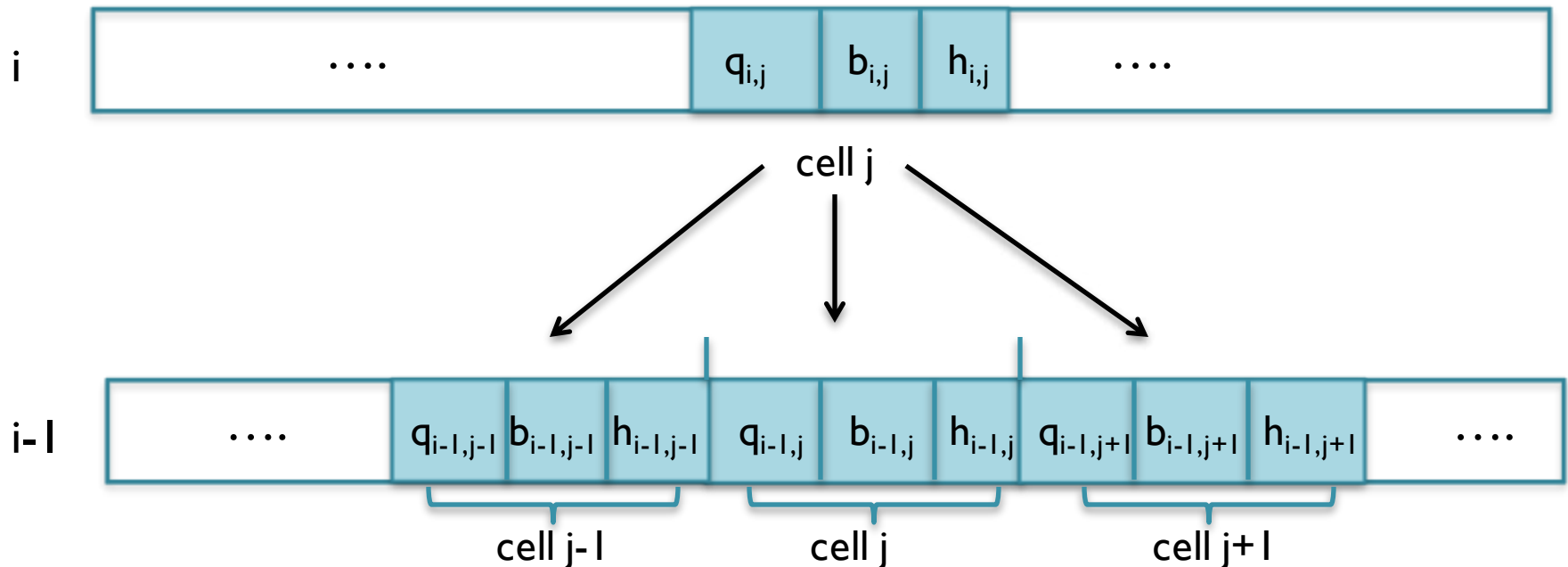
i



cell j

Main theorem: Step 2

- **Locality of computation:** The variables $h_{i,j}$, $b_{i,j}$ and $q_{i,j}$ depend only on the variables
 - $h_{i-1,j-1}$, $b_{i-1,j-1}$, $q_{i-1,j-1}$,
 - $h_{i-1,j}$, $b_{i-1,j}$, $q_{i-1,j}$, and
 - $h_{i-1,j+1}$, $b_{i-1,j+1}$, $q_{i-1,j+1}$



Main theorem: Step 2

- Hence,

$$b_{ij} = B_{ij}(h_{i-1,j-1}, b_{i-1,j-1}, q_{i-1,j-1}, h_{i-1,j}, b_{i-1,j}, q_{i-1,j}, h_{i-1,j+1}, b_{i-1,j+1}, q_{i-1,j+1})$$

= a fixed function of the arguments depending only on N 's transition function δ .

- The above equality can be captured by a constant size CNF Ψ_{ij} . Also, Ψ_{ij} is easily computable from δ .

Main theorem: Step 2


- Similarly,

$$h_{ij} = H_{ij}(h_{i-1,j-1}, b_{i-1,j-1}, q_{i-1,j-1}, h_{i-1,j}, b_{i-1,j}, q_{i-1,j}, h_{i-1,j+1}, b_{i-1,j+1}, q_{i-1,j+1})$$

= a fixed function of the arguments depending only on N 's transition function δ .

- The above equality can be captured by a constant size CNF Φ_{ij} . Also, Φ_{ij} is easily computable from δ .

Main theorem: Step 2

- Similarly,  k-th bit of q_{ij} where $1 \leq k \leq \log |Q|$
 $q_{ijk} = C_{ijk}(h_{i-1,j-1}, b_{i-1,j-1}, q_{i-1,j-1}, h_{i-1,j}, b_{i-1,j}, q_{i-1,j}, h_{i-1,j+1}, b_{i-1,j+1}, q_{i-1,j+1})$
= a fixed function of the arguments depending only
on N 's transition function δ .
- The above equality can be captured by a constant size CNF θ_{ijk} . Also, θ_{ijk} is easily computable from δ .

Main theorem: Step 2

- Let λ be the conjunction of Ψ_{ij} , Φ_{ij} and θ_{ijk} for all i, j, k .
 - $i \in [1, T(n)]$,
 - $j \in [1, T(n)]$, and
 - $k \in [1, \log |Q|]$
- λ is a CNF in the u -variables and the auxiliary variables.
Size of λ is $O(T(n)^2)$.

Main theorem: Step 2

- Let λ be the conjunction of Ψ_{ij} , Φ_{ij} and θ_{ijk} for all i, j, k .
 - $i \in [1, T(n)]$,
 - $j \in [1, T(n)]$, and
 - $k \in [1, \log |Q|]$
- λ is a CNF in the u -variables and the auxiliary variables.
Size of λ is $O(T(n)^2)$.
- Define $\phi = \lambda \wedge b_{T(n), 1}$

Main theorem: Step 2

- **Observe:** An assignment to u and the auxiliary variables satisfies λ if and only if it “captures” computation of N on the assigned input u and $N(u) = I$.

Main theorem: Step 2

- **Observe:** An assignment to u and the auxiliary variables satisfies λ if and only if it “captures” computation of N on the assigned input u and $N(u) = I$.
- Hence, an assignment to u and the auxiliary variables satisfies ϕ if and only if $N(u) = I$.

$$\phi(u, \text{“auxiliary variables”}) \in \text{SAT} \iff N(u) = I.$$

Main theorem: Comments

- ϕ is a CNF of size $O(T(n)^2)$ and is also computable from N and n in $O(T(n)^2)$ time.
- **Remark 1.** With some more effort, size ϕ can be brought down to $O(T(n) \cdot \log T(n))$.
- **Remark 2.** The reduction from N, u to $\phi(u, \dots)$ is not just a poly-time reduction, it is actually a log-space reduction (we'll define this later).

Main theorem: Comments

- ϕ is a function of u (the input) and some “auxiliary variables” (the b_{ij} , h_{ij} and q_{ijk} variables).
- Observe that once u is fixed the values of the “auxiliary variables” are also determined in any satisfying assignment for ϕ .
- Each clause of ϕ has only constantly many literals!

3SAT is NP-complete

- **Definition.** A CNF is called a **k-CNF** if every clause has at most **k** literals.

e.g. a 2-CNF $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- **Definition.** **k-SAT** is the language consisting of all *satisfiable k-CNFs*.

3SAT is NP-complete

- **Definition.** A CNF is called a **k-CNF** if every clause has at most **k** literals.

e.g. a 2-CNF $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- **Definition.** **k-SAT** is the language consisting of all *satisfiable k-CNFs*.

- **Theorem.** **3-SAT** is NP-complete.

Proof sketch: $(x_1 \vee x_2 \vee x_3 \vee \neg x_4)$ is satisfiable iff $(x_1 \vee x_2 \vee z) \wedge (x_3 \vee \neg x_4 \vee \neg z)$ is satisfiable.