Computational Complexity Theory

Lecture 4: Cook-Levin theorem

Department of Computer Science,
Indian Institute of Science

Recap: 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 $x \in L$.

Recap: 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
- 0/1 integer programming
- Integer factoring
- Graph isomorphism
- 2-Diophantine solvability

Recap: Is P = NP ?

- Obviously, $P \subseteq NP$.

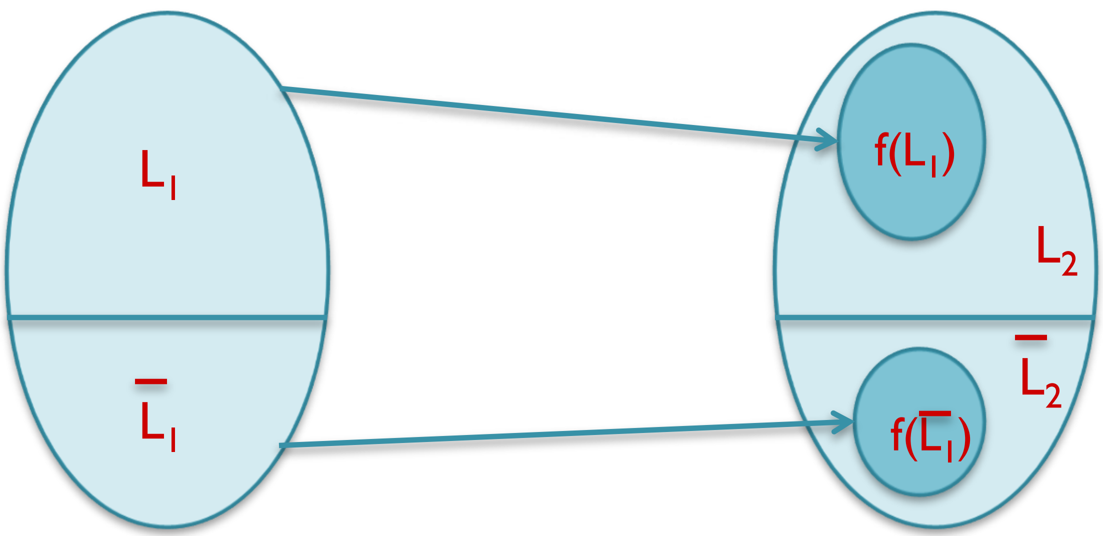
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!

- Solving a problem does seem harder than verifying its solution, so most people believe that $P \neq NP$.

Recap: 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.

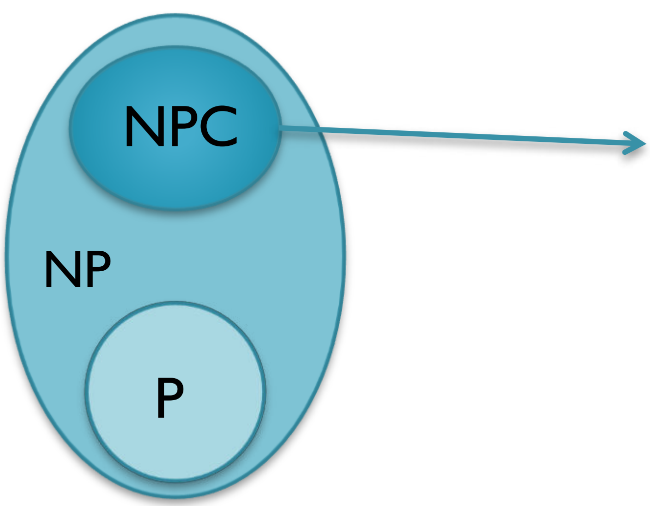
$$x \in L_1 \Leftrightarrow f(x) \in L_2$$



Recap: 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.



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

Recap: 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)

How to show existence of an NPC problem?

- Let L' = { $(\alpha, x, 1^m, 1^t)$: there exists a $u \in \{0,1\}^m$ s.t. M_α accepts (x, u) in t 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 $x_1, \ldots, x_n$ 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. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

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

clauses

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 ∈ NP. We intend to come up with a polynomial-time computable function f: $x \longmapsto \phi_x$ s.t.,
  - $x \in L \Leftrightarrow \phi_x \in SAT$

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 $f: x \longmapsto \phi_x$ 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

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

Cook-Levin theorem: Proof

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

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

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

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

Cook-Levin theorem: Proof

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

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

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

  $\exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x, u) = 1 \Leftrightarrow \phi_x$ 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, (think of $N = M(x, ..)$ for a fixed x.)

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 ϕ(u, "*auxiliary variables*") of size *poly*(T(n)) such that for every u, ϕ(u, "*auxiliary variables*") is satisfiable as a function of the "*auxiliary variables*" if and only if N(u) = 1.
  2. ϕ is computable in time *poly(T(n))* from N, T & 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 ϕ(u, "*auxiliary variables*") of size *poly*(T(n)) such that for every u, ϕ(u, "*auxiliary variables*") is satisfiable as a function of the "*auxiliary variables*" if and only if N(u) = 1.
  2. ϕ is computable in time *poly(T(n))* from N, T & n.

- ϕ(u, "*auxiliary variables*") is satisfiable as a function of **all** the variables if and only if ∃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 ϕ(u, "*auxiliary variables*") of size *poly*(T(n)) such that for every u, ϕ(u, "*auxiliary variables*") is satisfiable as a function of the "*auxiliary variables*" if and only if N(u) =1.
  2. ϕ is computable in time *poly(T(n))* from N, T & 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 *poly*$(T(n))$ such that $\psi(u) = 1$ if and only if $N(u) = 1$.
  2. ψ is computable in time *poly(T(n))* from N, T & n.

- Step 2. "Convert" circuit ψ to a CNF ϕ efficiently by introducing auxiliary variables.

Main theorem: Proof

- <u>**Step 1**</u>. 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 *poly*(T(n)) such that ψ(u) = 1 if and only if N(u) =1.
  2. ψ is computable in time *poly(T(n))* from N, T & n.

The key insight: ψ "encodes" N.

- Step 2. "Convert" circuit ψ to a CNF ϕ efficiently by introducing <u>auxiliary variables</u>.

Main theorem: Step 1

- 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 1

- 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

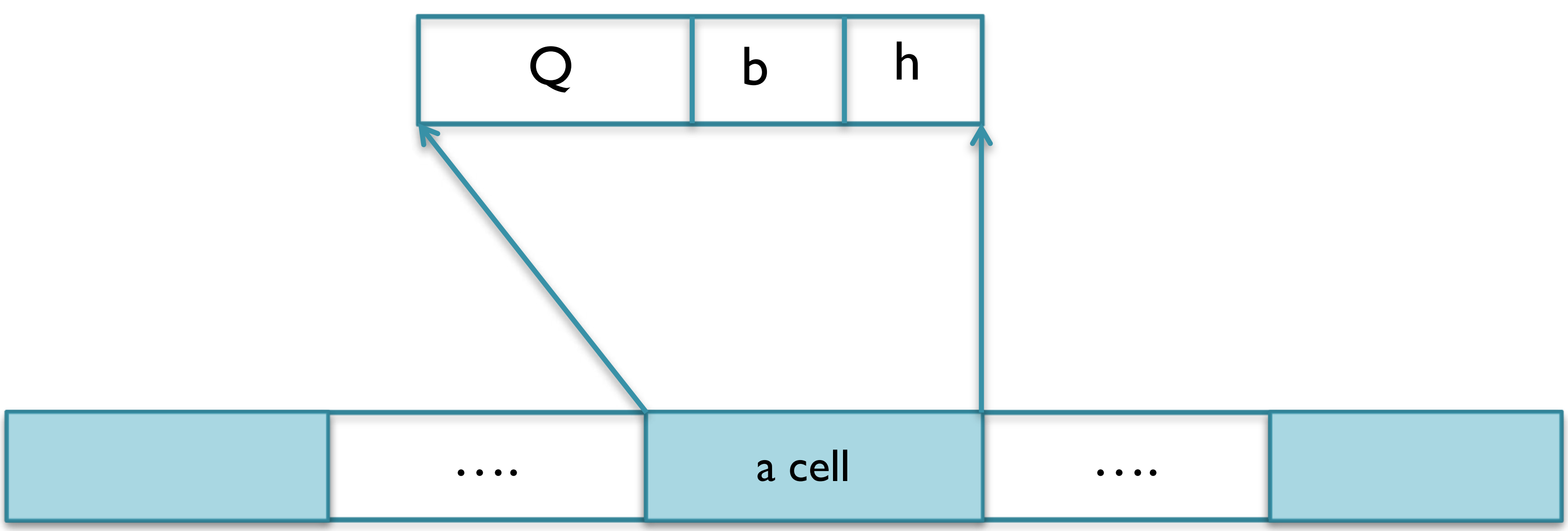
Main theorem: Step 1

- 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

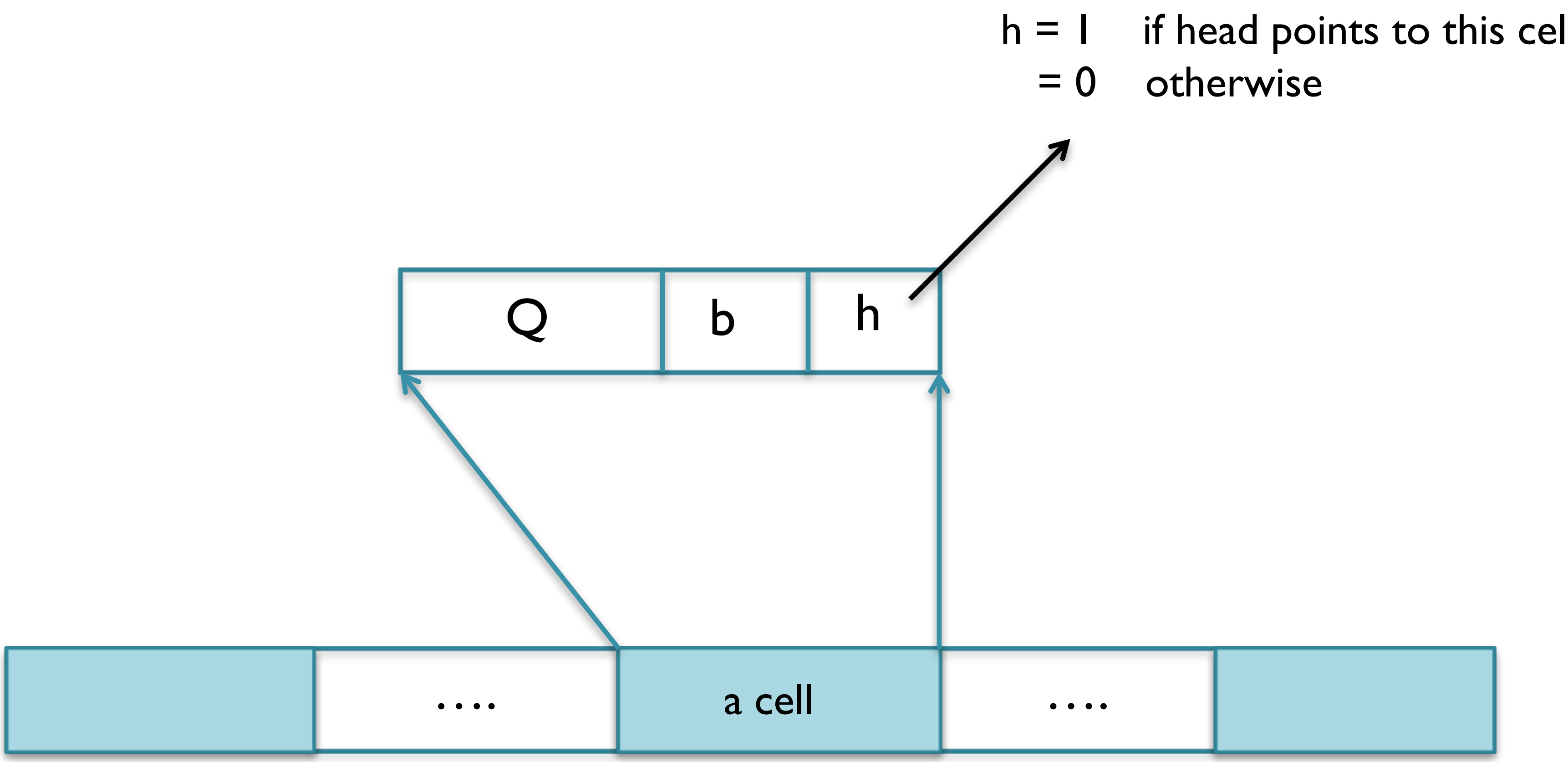
- Think of a '<u>compound</u>' tape: Every cell stores the current state, a bit content and head indicator.

Main theorem: Step 1



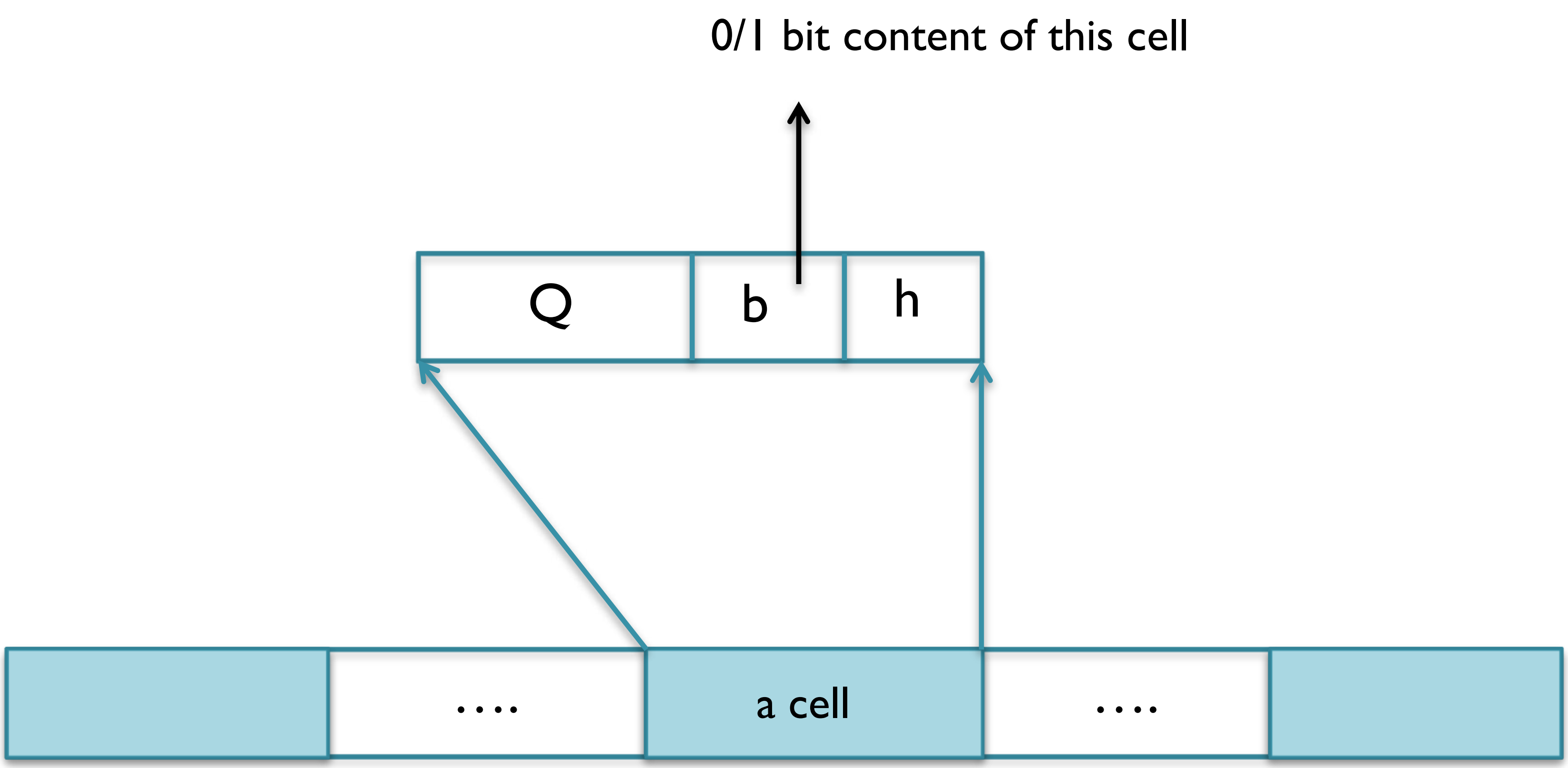
A compound tape

Main theorem: Step 1



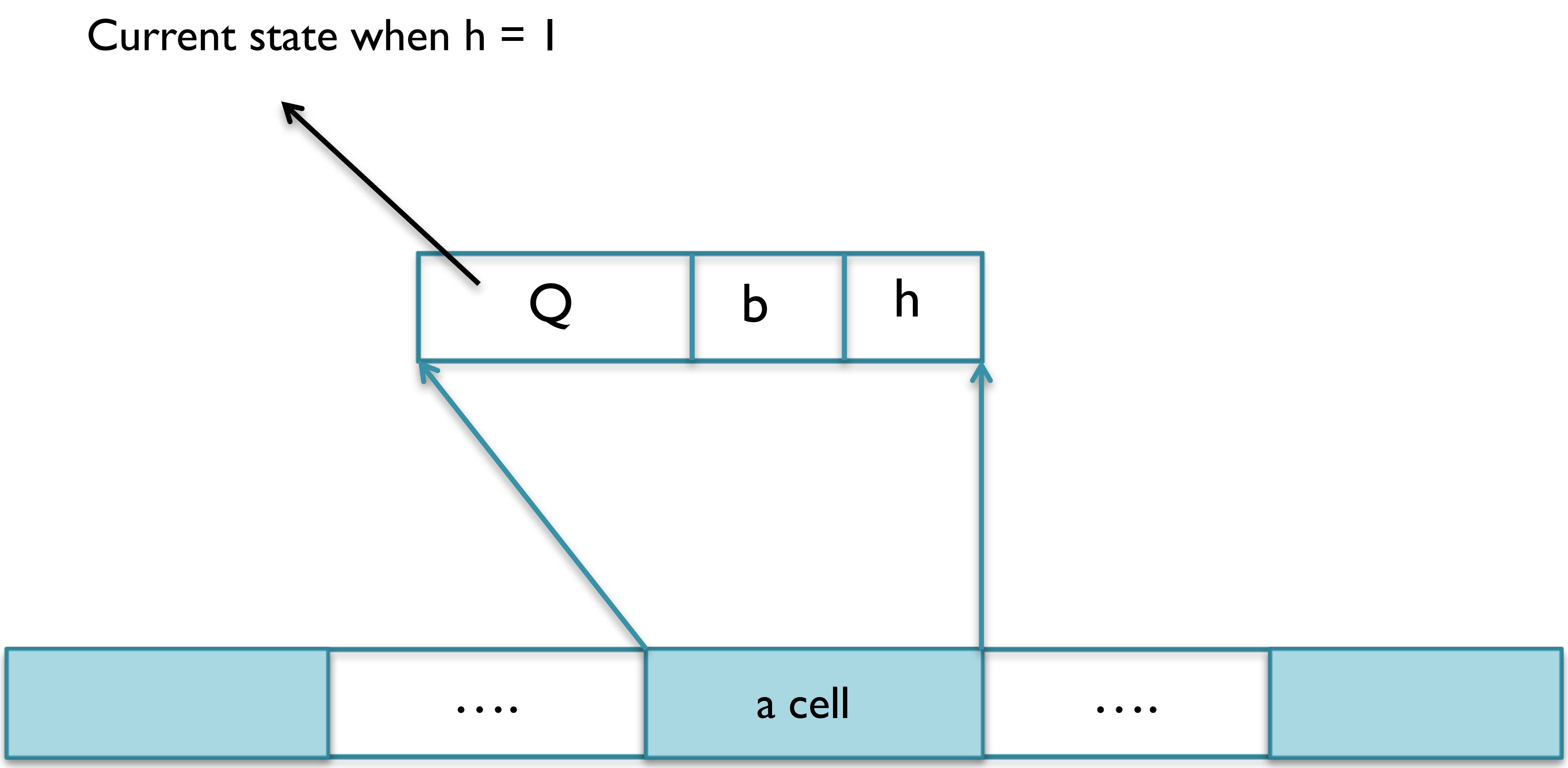
A compound tape

Main theorem: Step 1



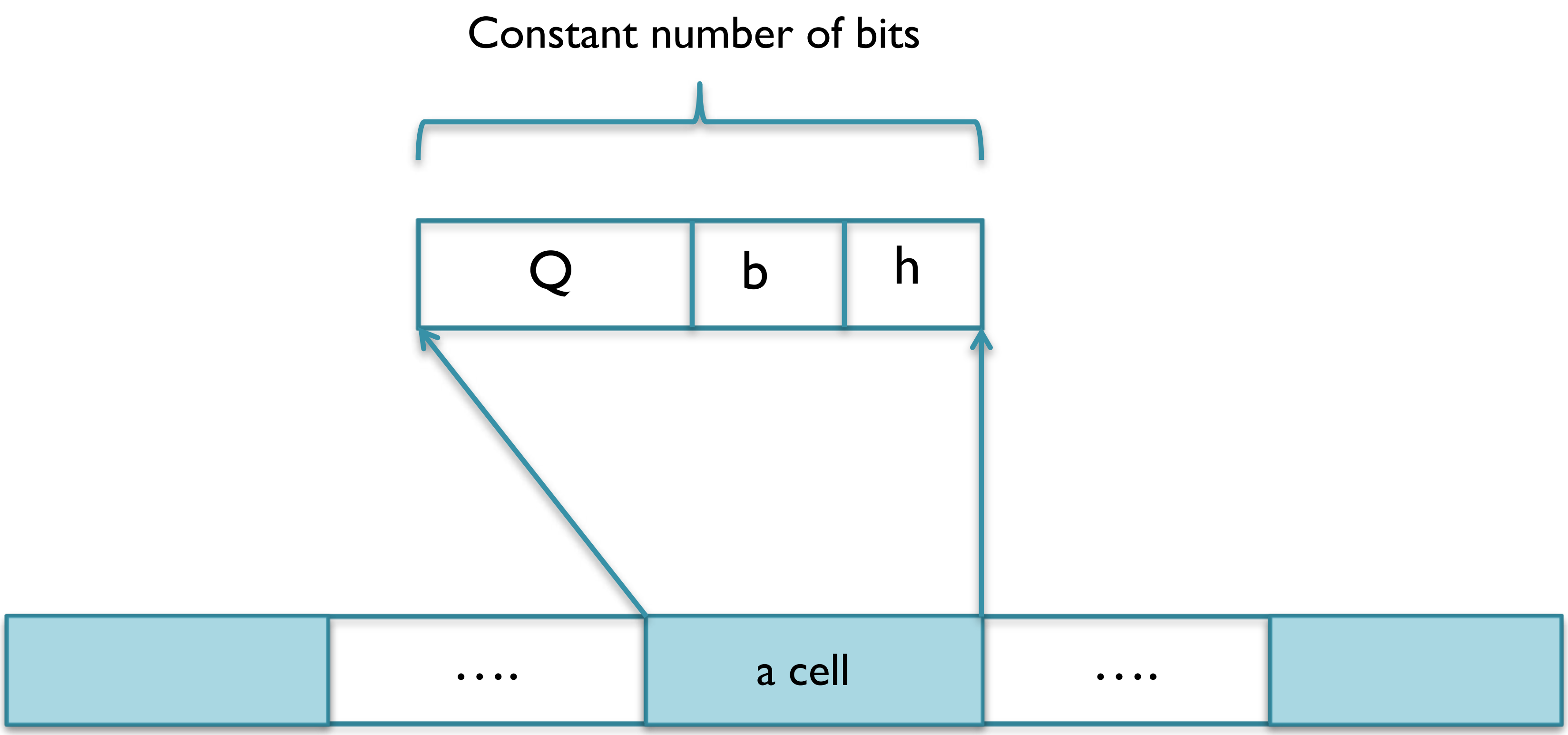
A compound tape

Main theorem: Step 1



A compound tape

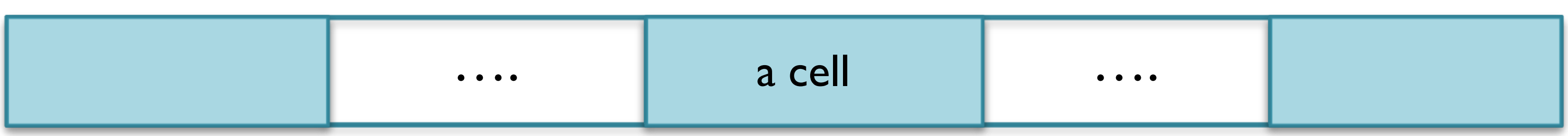
Main theorem: Step 1



A compound tape

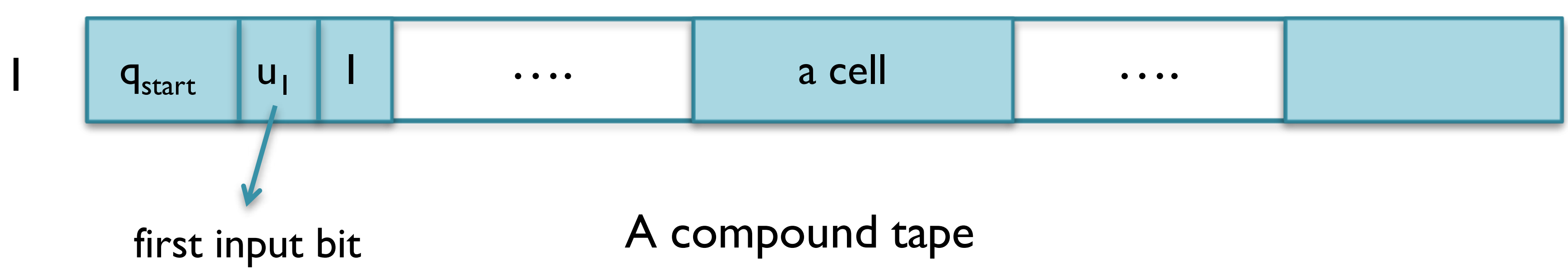
Main theorem: Step 1

- Computation of N on inputs of length n can be <u>completely described</u> 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.

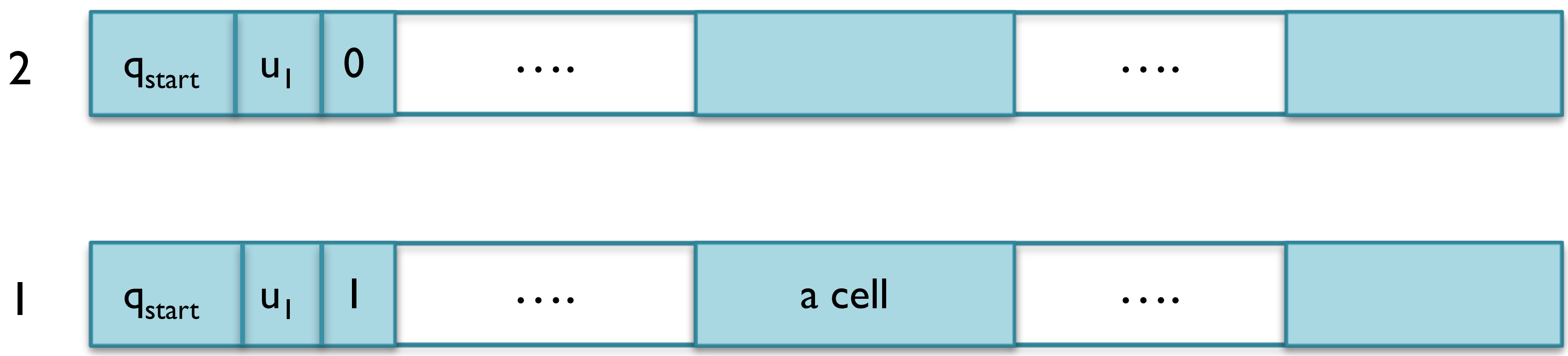


A compound tape

Main theorem: Step 1

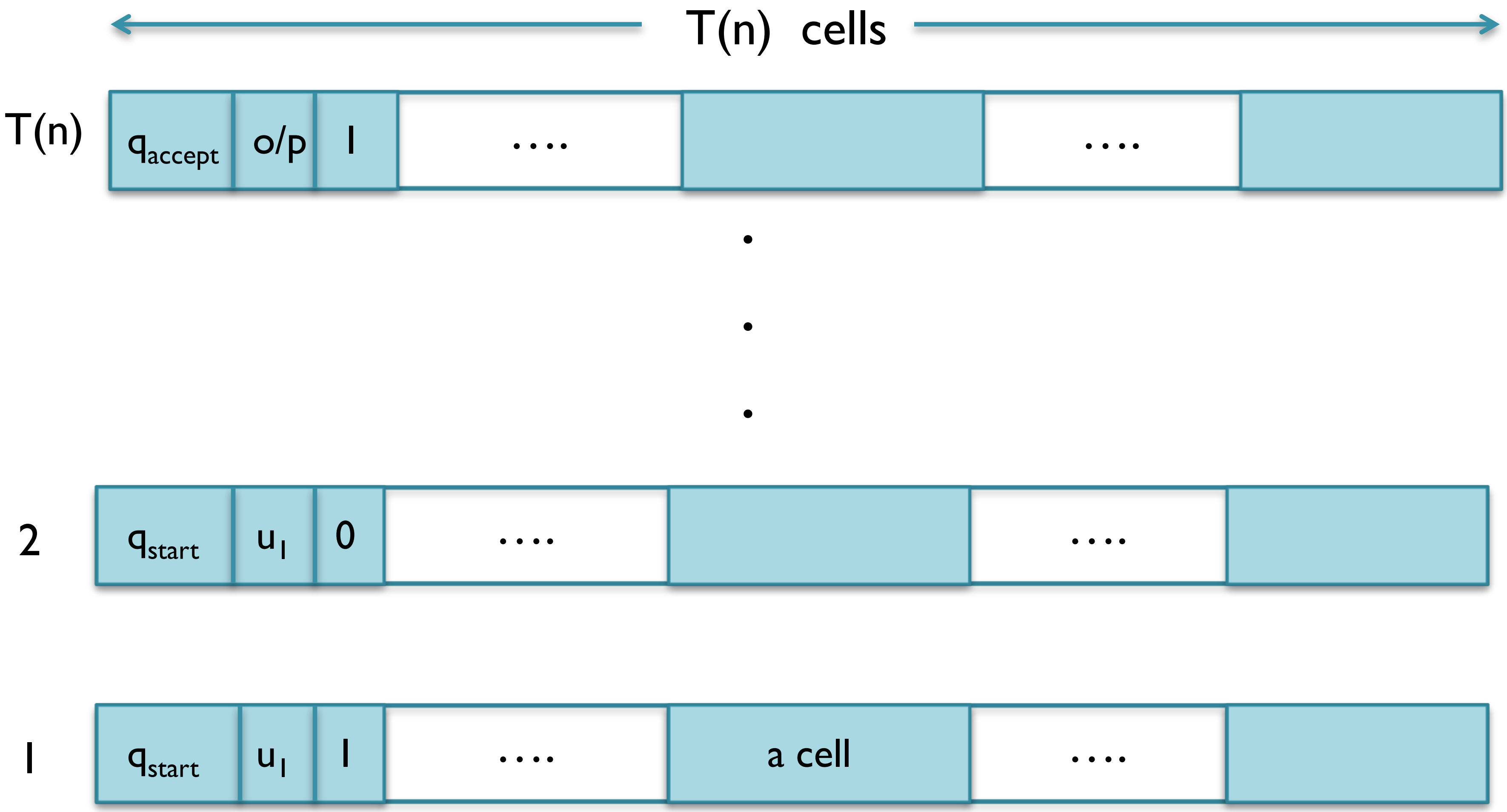


Main theorem: Step 1



A compound tape

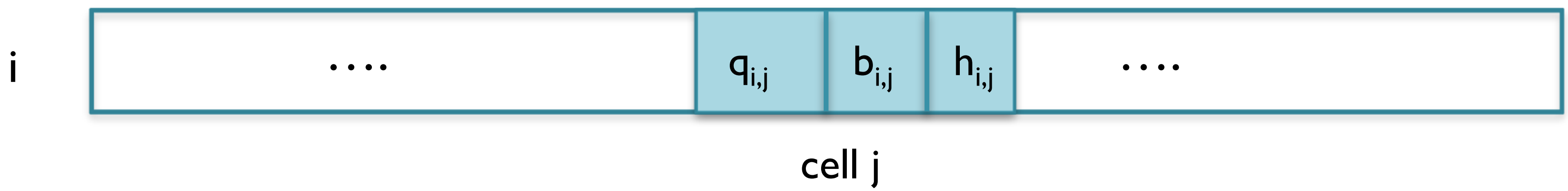
Main theorem: Step 1



A compound tape

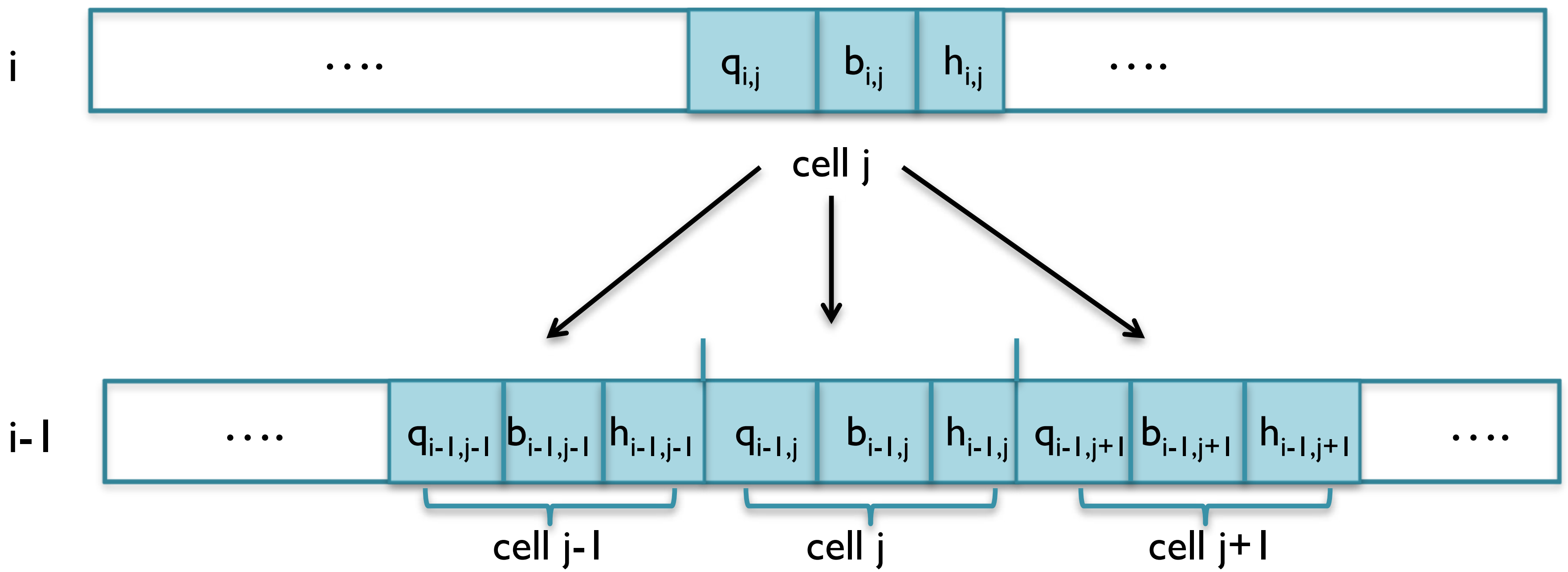
Main theorem: Step 1

- $h_{i,j}$ = 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



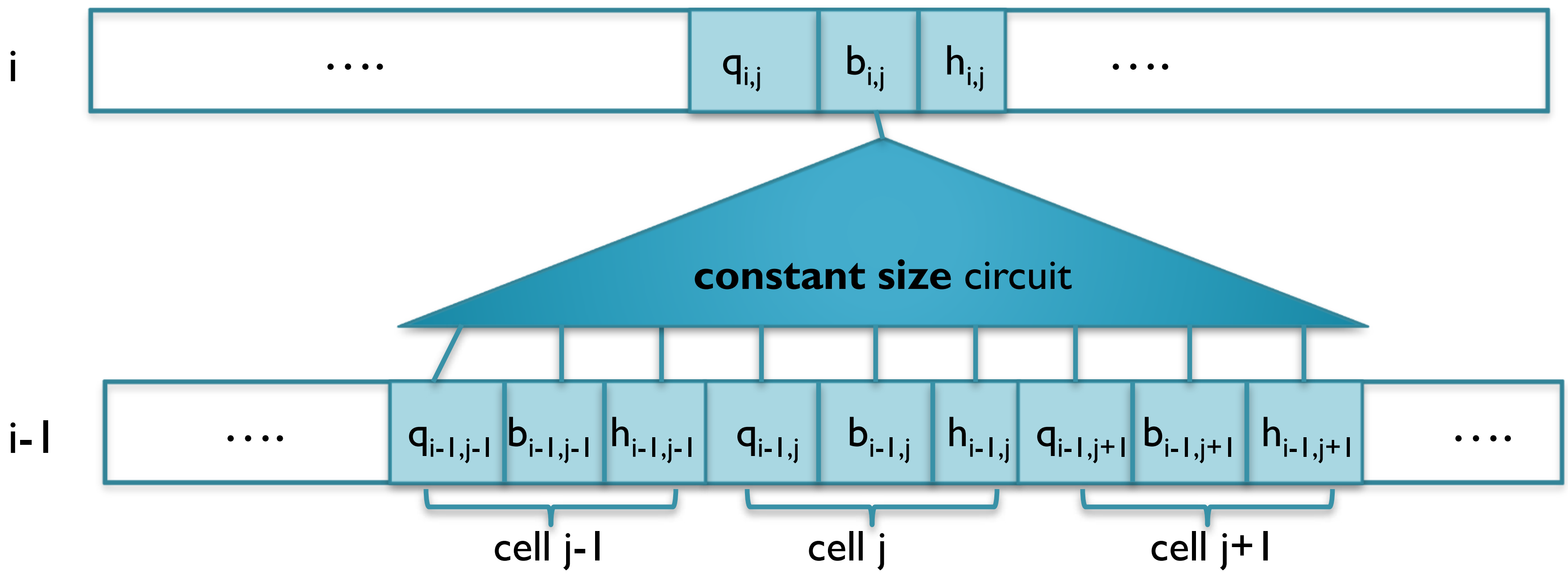
Main theorem: Step 1

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

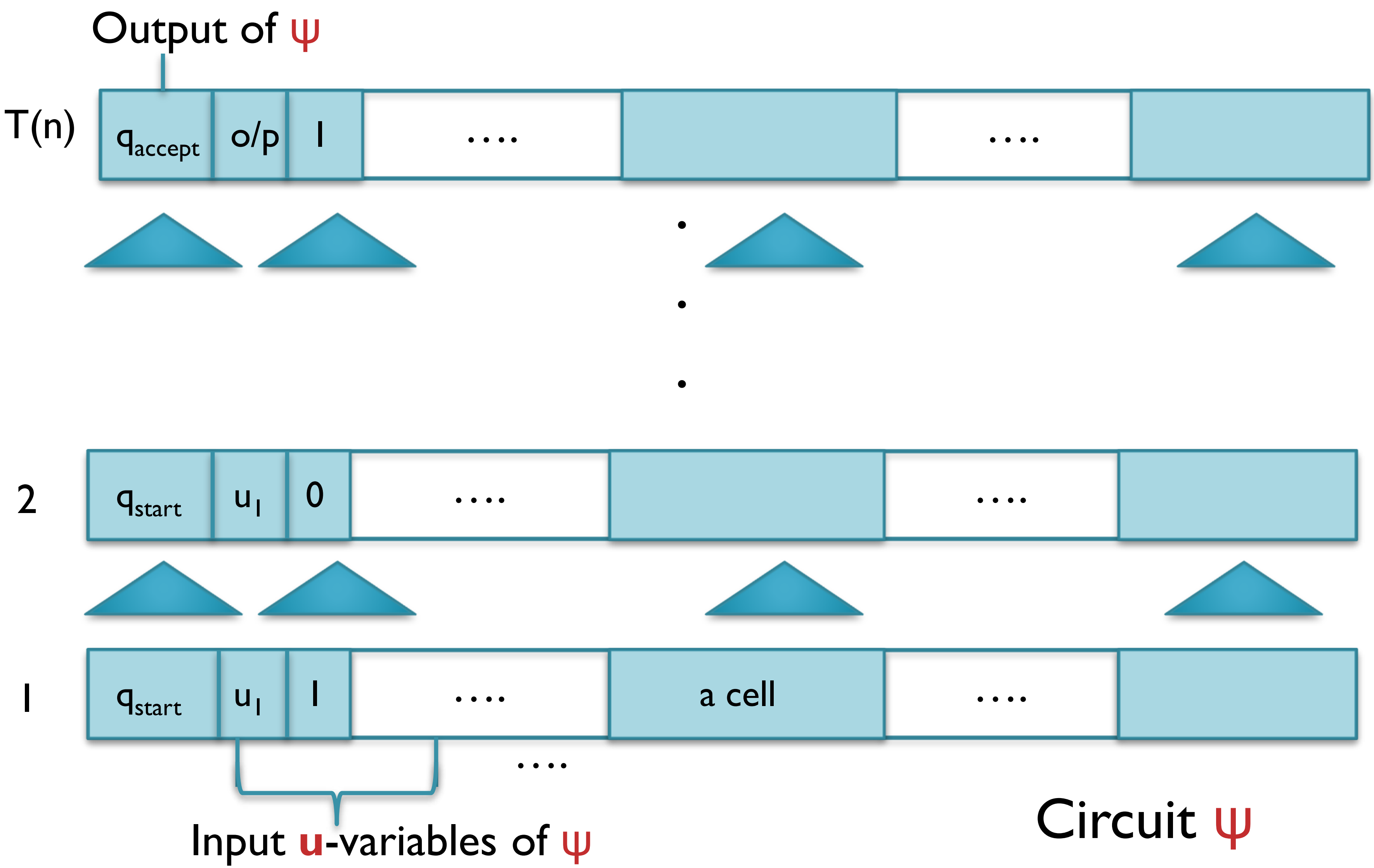


Main theorem: Step 1

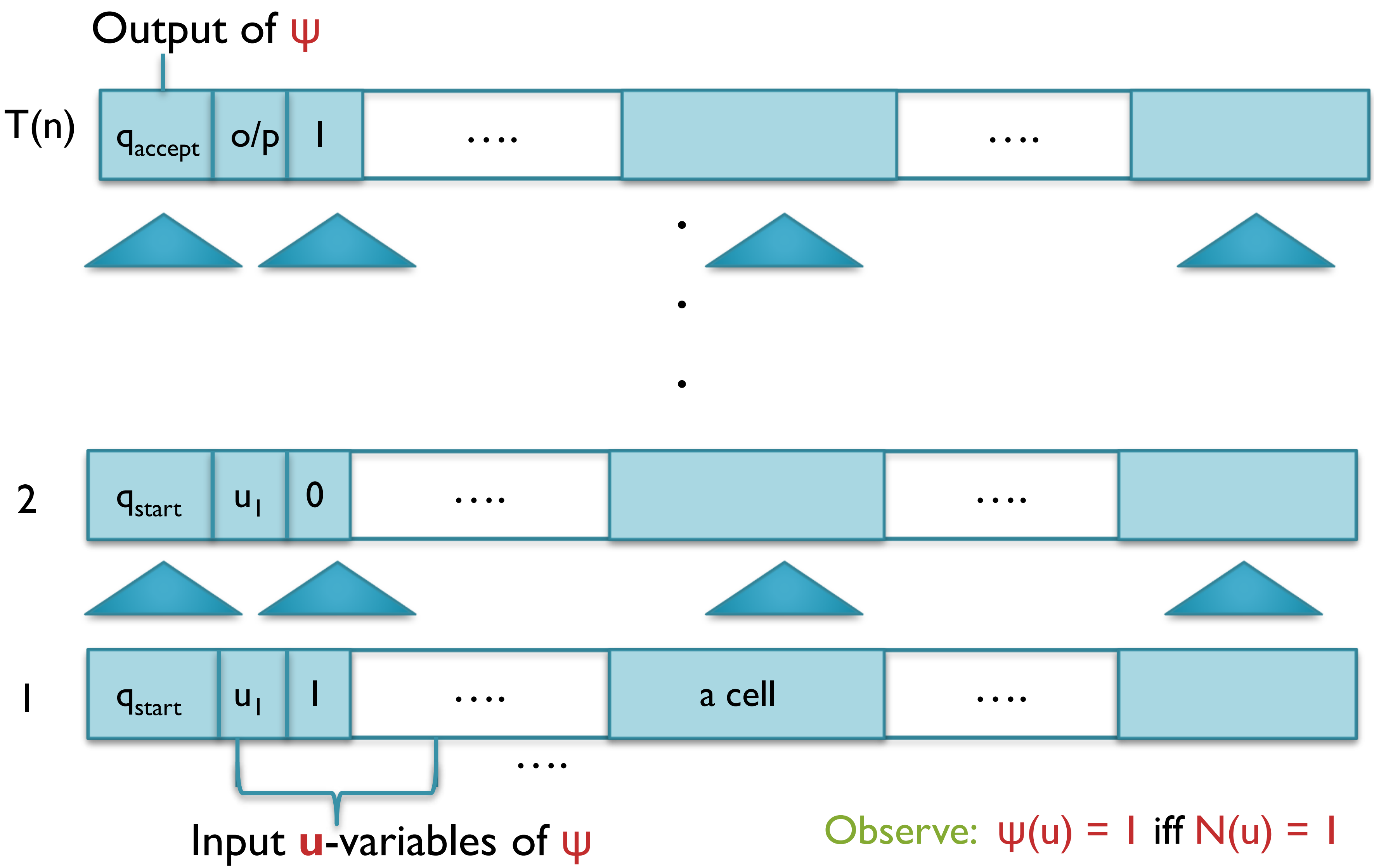
- **Locality of computation:** The bits in $h_{i,j}$, $b_{i,j}$ and $q_{i,j}$ depend ***only on*** the bits in
  - $h_{i-1,j-1}$, $b_{i-1,j-1}$, $q_{i-1,j-1}$,
  - $h_{i-1,j}$, $b_{i-1,j}$, $q_{i-1,j}$,
  - $h_{i-1,j+1}$, $b_{i-1,j+1}$, $q_{i-1,j+1}$



Main theorem: Step 1



Main theorem: Step 1



Observe: ψ(u) = 1 iff N(u) = 1

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 *poly*(T(n)) such that ψ(u) = 1 if and only if N(u) = 1.
    2. ψ is computable in time *poly(T(n))* from N, T & n.

- **Step 2.** "Convert" circuit ψ to a CNF ϕ efficiently by introducing auxiliary variables.