Computational Complexity Theory

Lecture 3: Reductions; NP-completeness; Cook-Levin theorem

Department of Computer Science,
Indian Institute of Science

Recap: Decision Problems

Decision problems ⟺ Boolean functions ⟺ Languages

- Definition. We say a TM M *decides* a language $L \subseteq \{0,1\}^*$ if M computes f_L, where $f_L(x) = 1$ if and only if $x \in L$.

The characteristic function of L .

Recap: Complexity Class P

- Let $T: \mathbb{N} \rightarrow \mathbb{N}$ be some function.
- Definition: A language L is in $DTIME(T(n))$ if there's a TM that decides L in time $O(T(n))$.
- Defintion: Class $P = \bigcup_{c > 0} DTIME(n^c)$.

Deterministic polynomial-time

Recap: Problems in P

- Cycle detection
- Solvabililty of a system of linear equations
- Perfect matching
- Planarity testing
- Primality testing

Recap: Polynomial-time TM

- Definition. A TM M is a *polynimial-time* TM if there's a polynomial function $q: \mathbb{N} \rightarrow \mathbb{N}$ such that for every input $x \in \{0,1\}^*$, M halts within $q(|x|)$ steps.

Polynomial function. $q(n) = O(n^c)$ for some constant c.

Recap: Class FP

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the i-th bit of the output and make it a decision problem.

- We say that a problem or a function $f: \{0,1\}^* \longrightarrow \{0,1\}^*$ is in FP (functional P) if there's a polynomial-time TM that computes f.

Complexity Class FP : Examples

- Greatest Common Divisor
- Counting paths in a DAG
- Maximum matching
- Linear Programming
- Factoring Polynomials

Recap: 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

Recap: 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: 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.

Recap: Problems in NP

- Vertex cover
- 0/1 integer programming
- Integer factoring
- Graph isomorphism
- 2-Diophantine solvability

Recap: Is P = NP ?

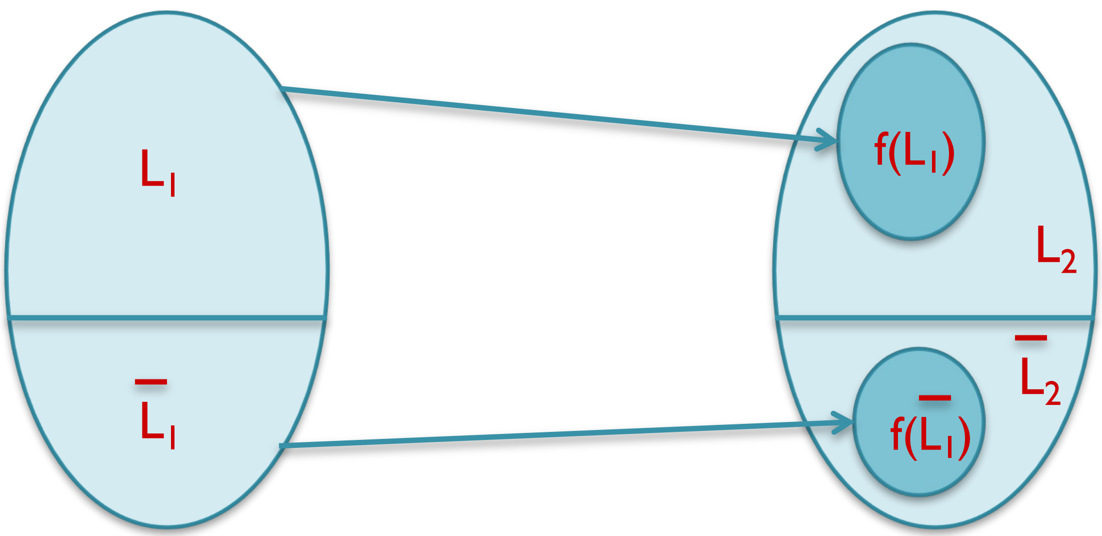
- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- Solving a problem does seem harder than verifying its solution, so most people believe that $P \neq NP$.

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.

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



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.

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

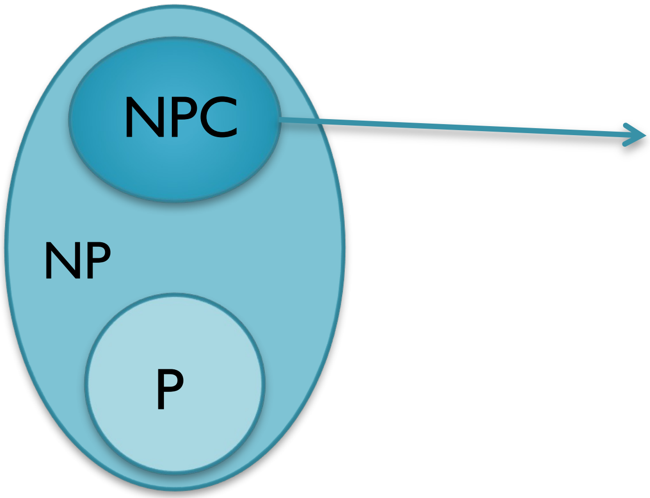
- Notation. $L_1 \leq_p L_2$

- Observe. If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.
  (*Transitivity*)

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.

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' = { $(\alpha, x, 1^m, 1^t)$: 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. $\varphi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- 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. $\varphi = (x_1 \lor x_2) \land (x_3 \lor \neg x_2)$

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

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

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

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

- 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 $f: x \longmapsto \varphi_x$ s.t.,
  - $x \in L \Leftrightarrow \varphi_x \in SAT$

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 $f: x \longmapsto \varphi_x$ s.t.,
    - $x \in L \Leftrightarrow \varphi_x \in SAT$

    - Notation: $|\varphi_x| :=$ 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

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

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 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 auxiliary variables.

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

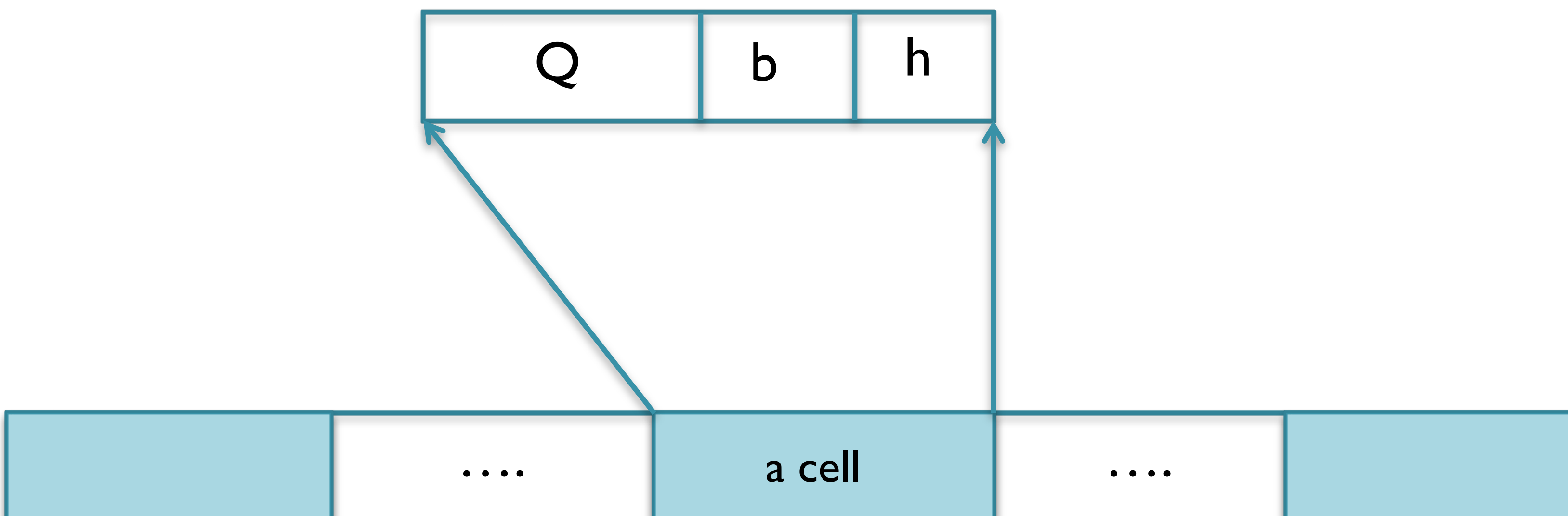
Main theorem: Step 1

- 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

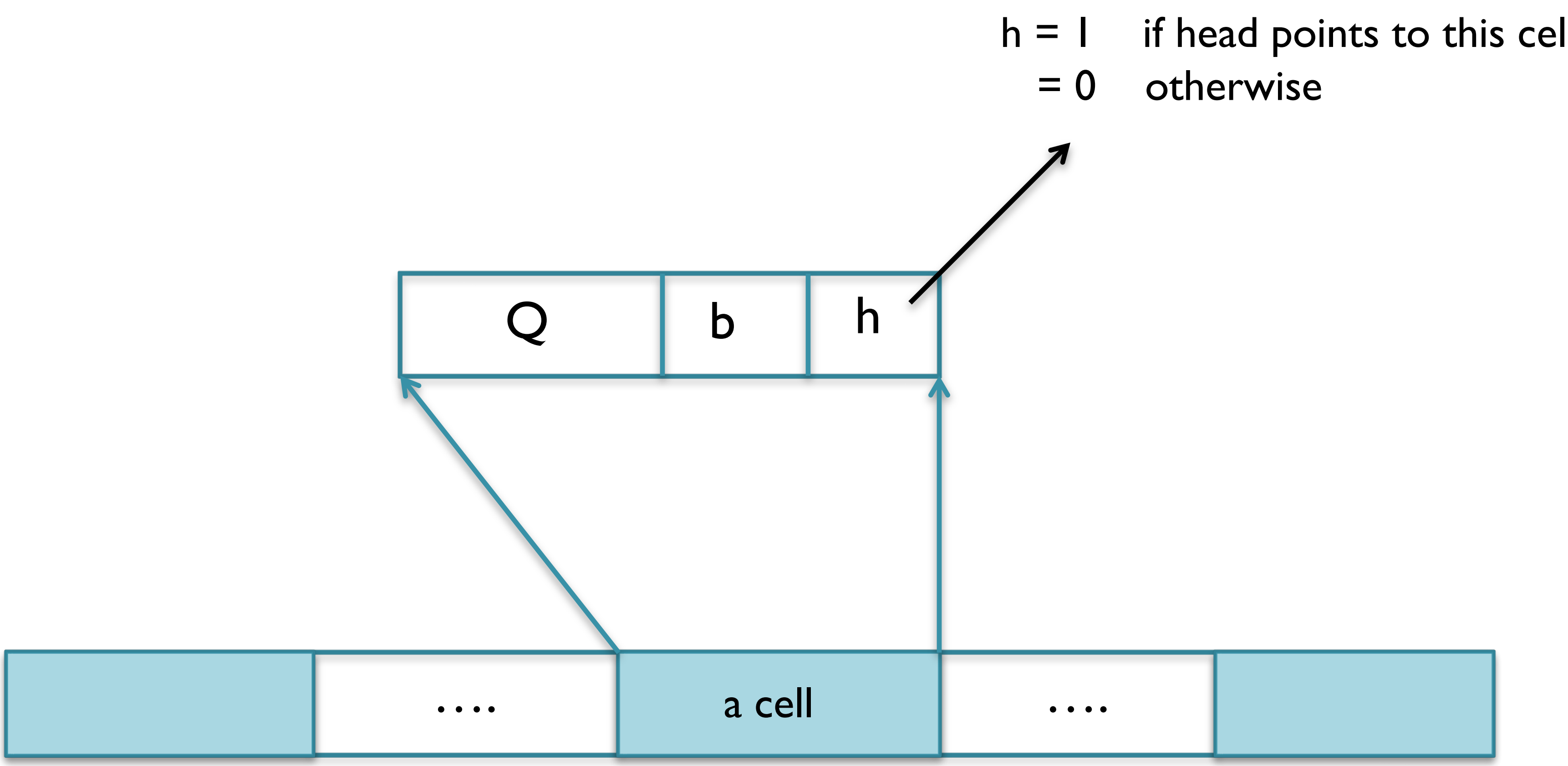
- Think of a ‘<u>compound</u>’ tape: Every cell stores the current state, a bit content and head indicator.

Main theorem: Step 1



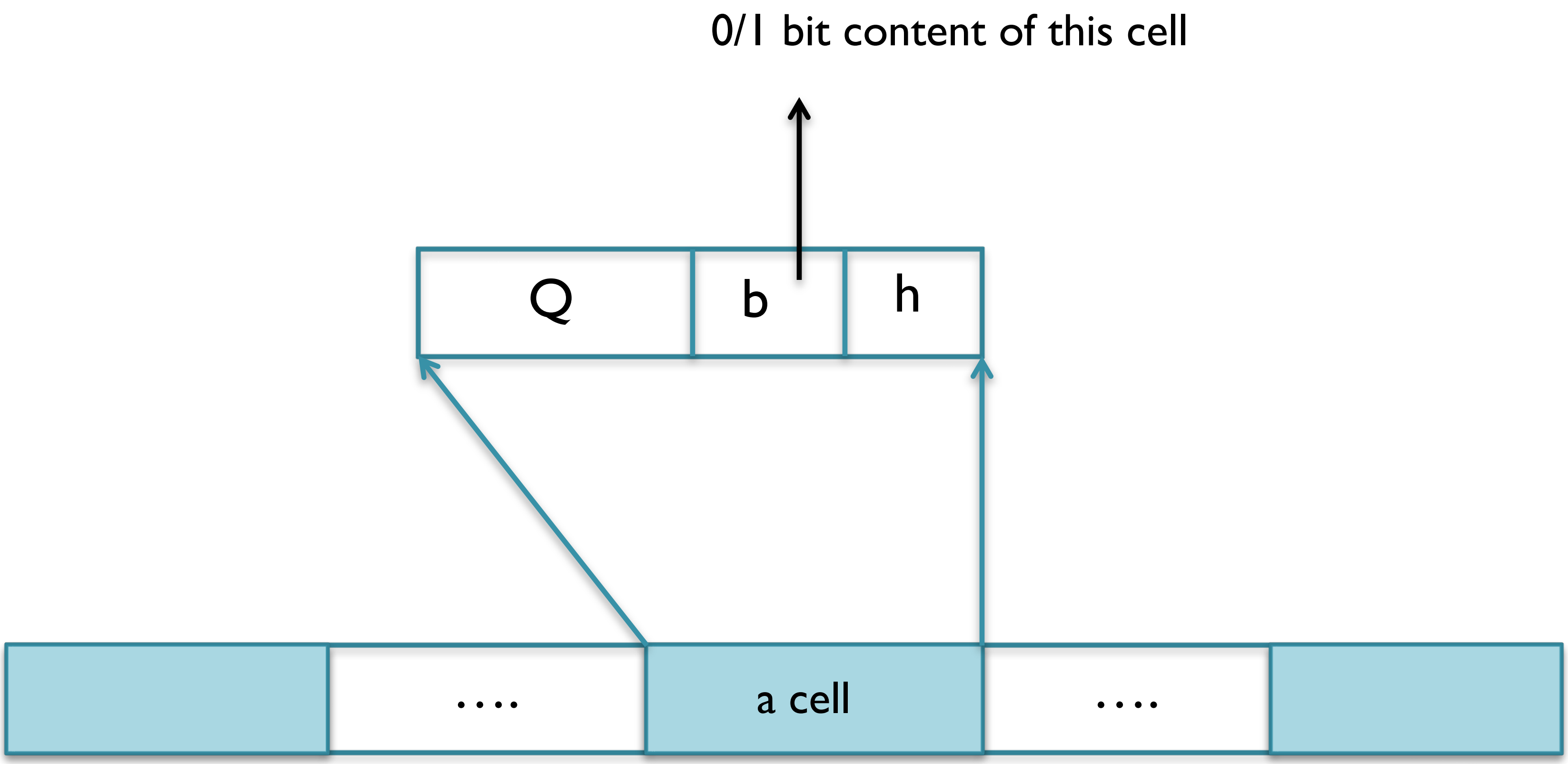
A compound tape

Main theorem: Step 1



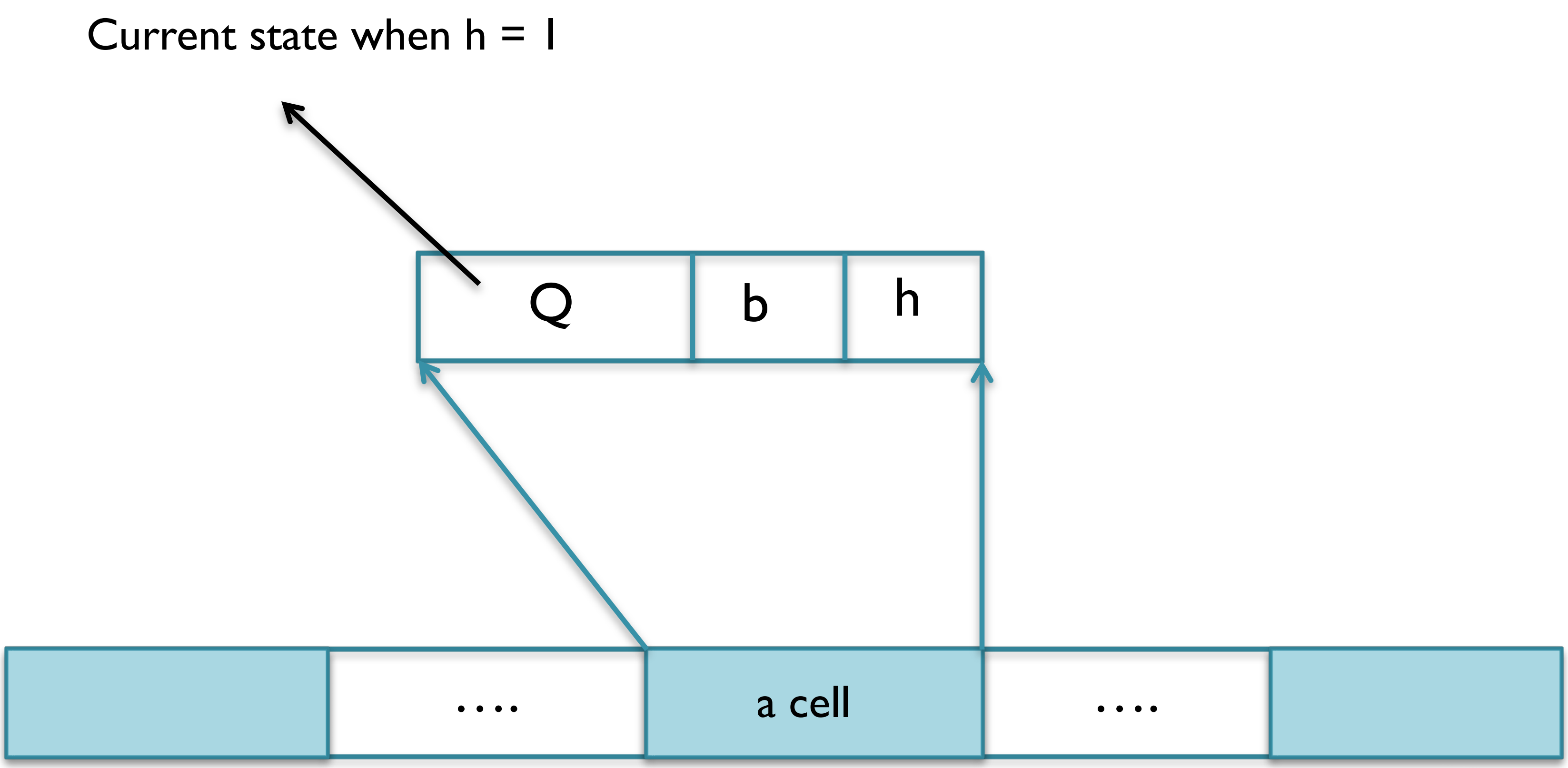
A compound tape

Main theorem: Step 1



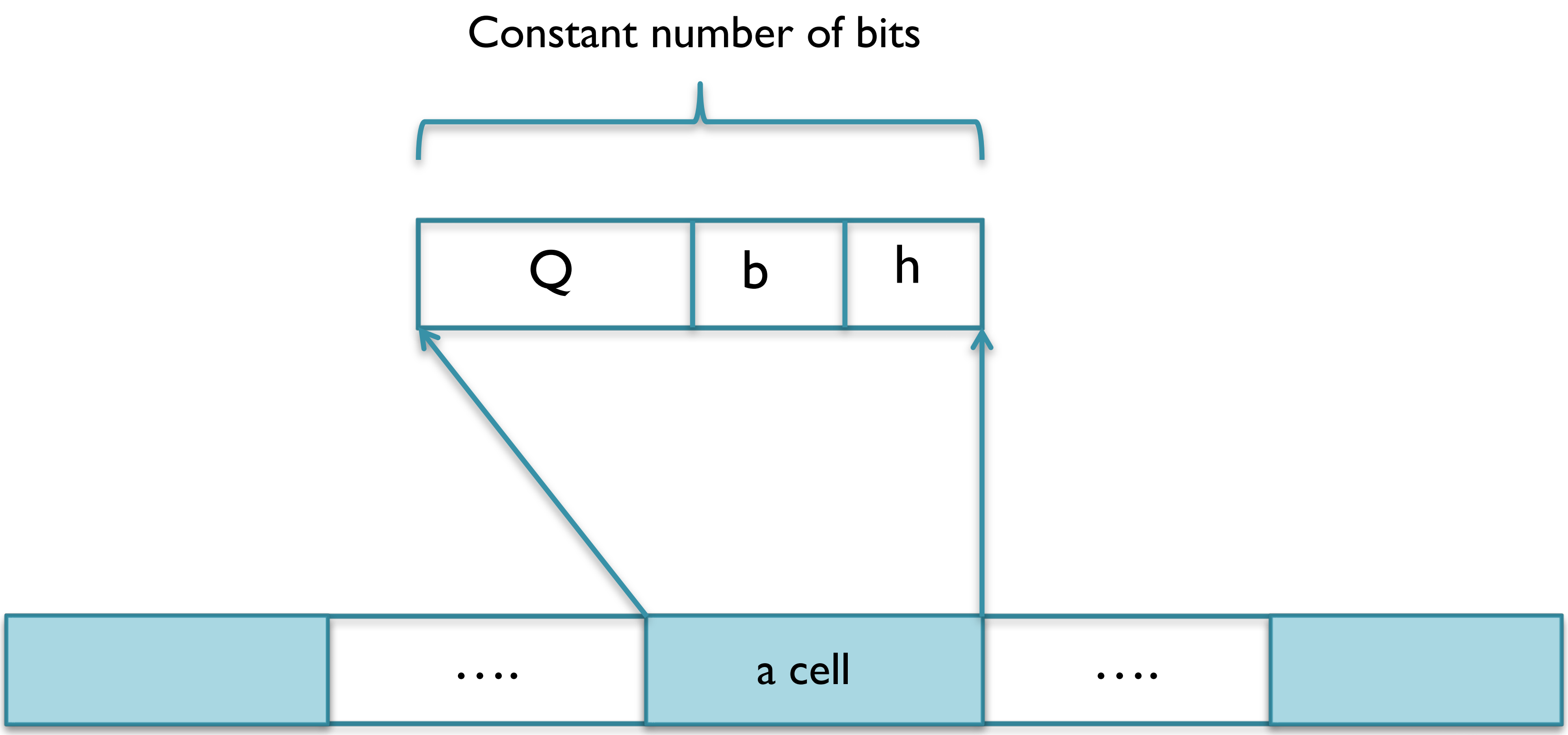
A compound tape

Main theorem: Step 1



A compound tape

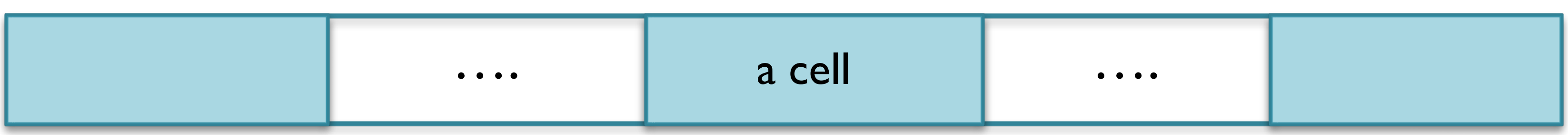
Main theorem: Step 1



A compound tape

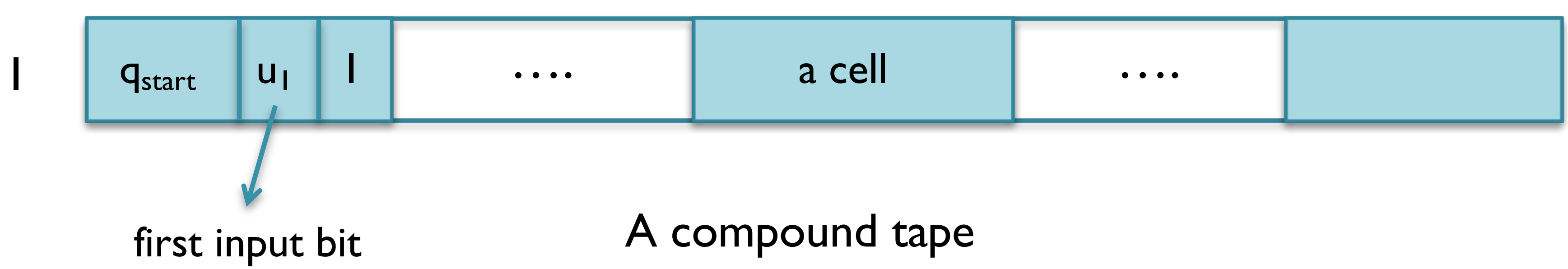
Main theorem: Step 1

- Computation of N on inputs of length n can be 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.

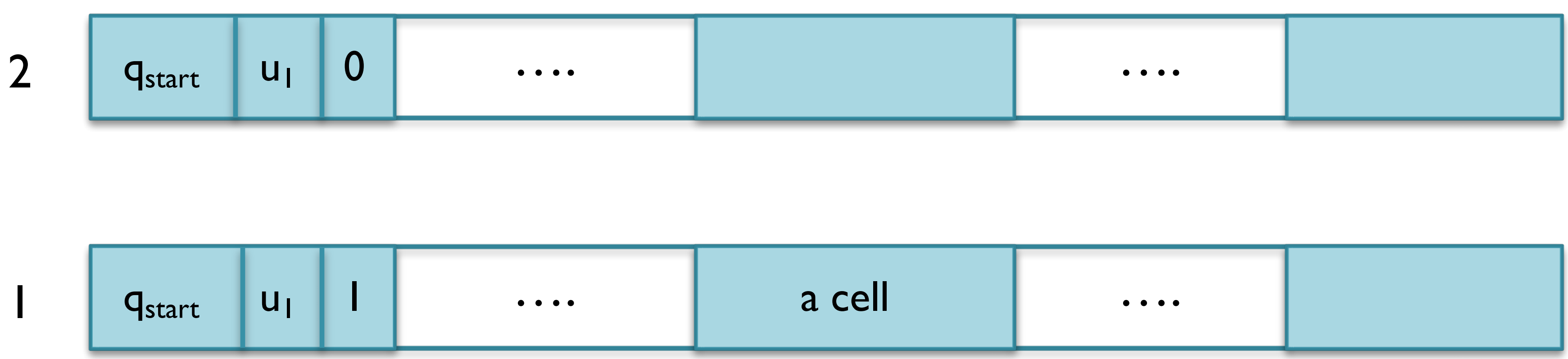


A compound tape

Main theorem: Step 1

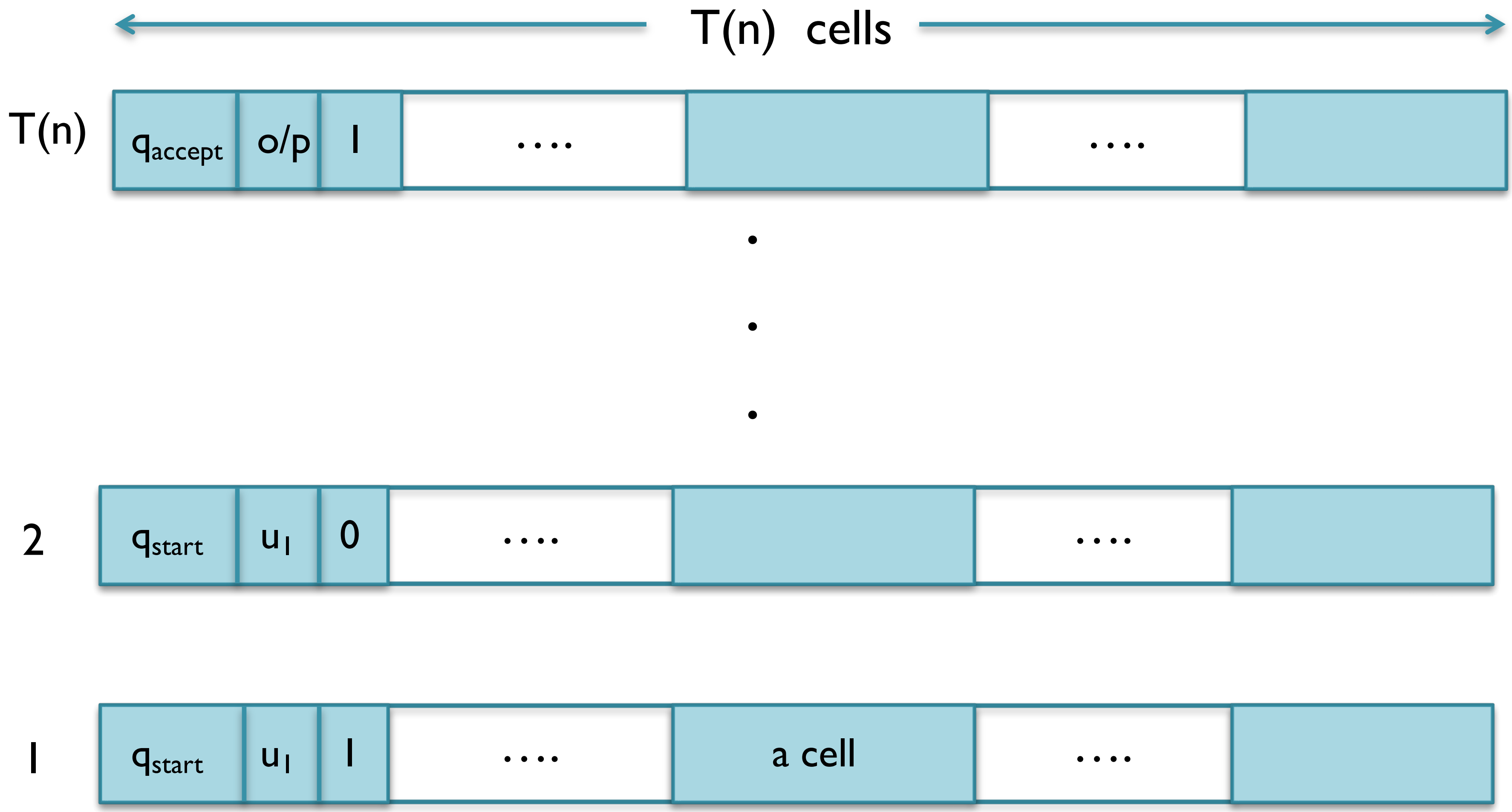


Main theorem: Step 1



A compound tape

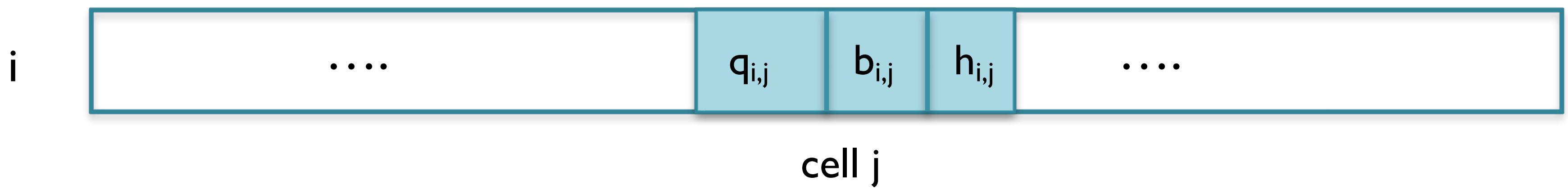
Main theorem: Step 1



A compound tape

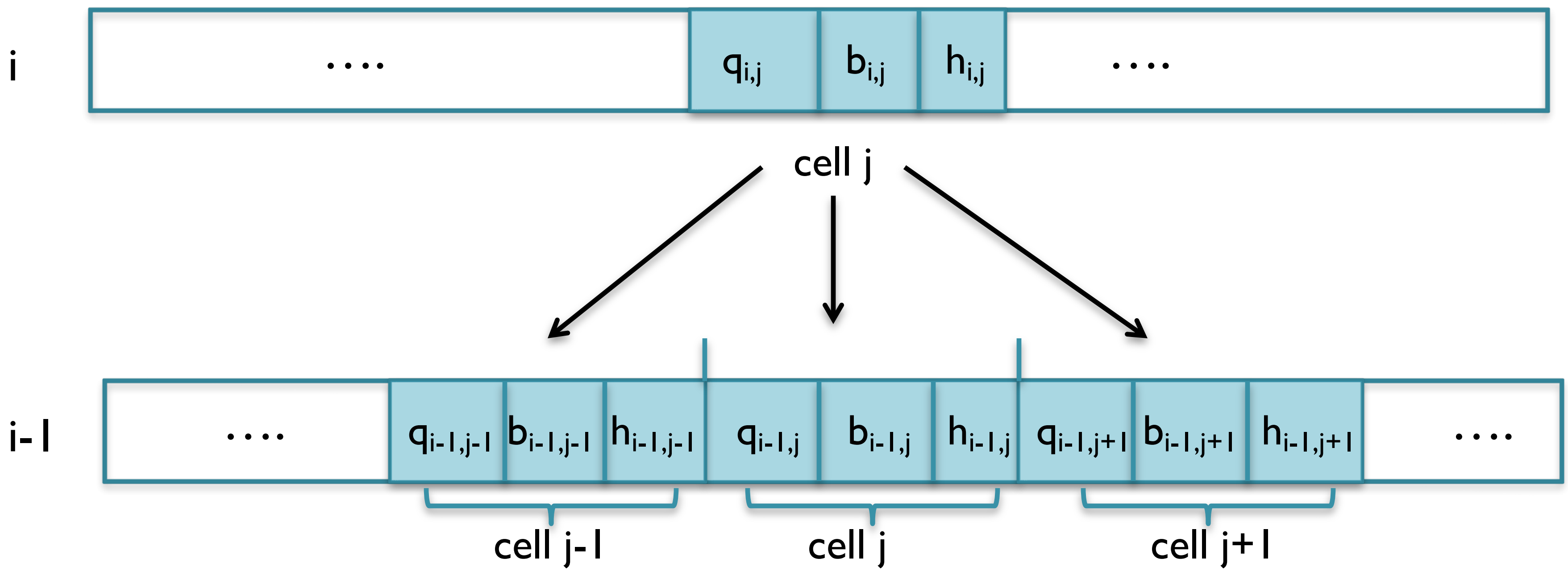
Main theorem: Step 1

- $h_{i,j}$ = 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



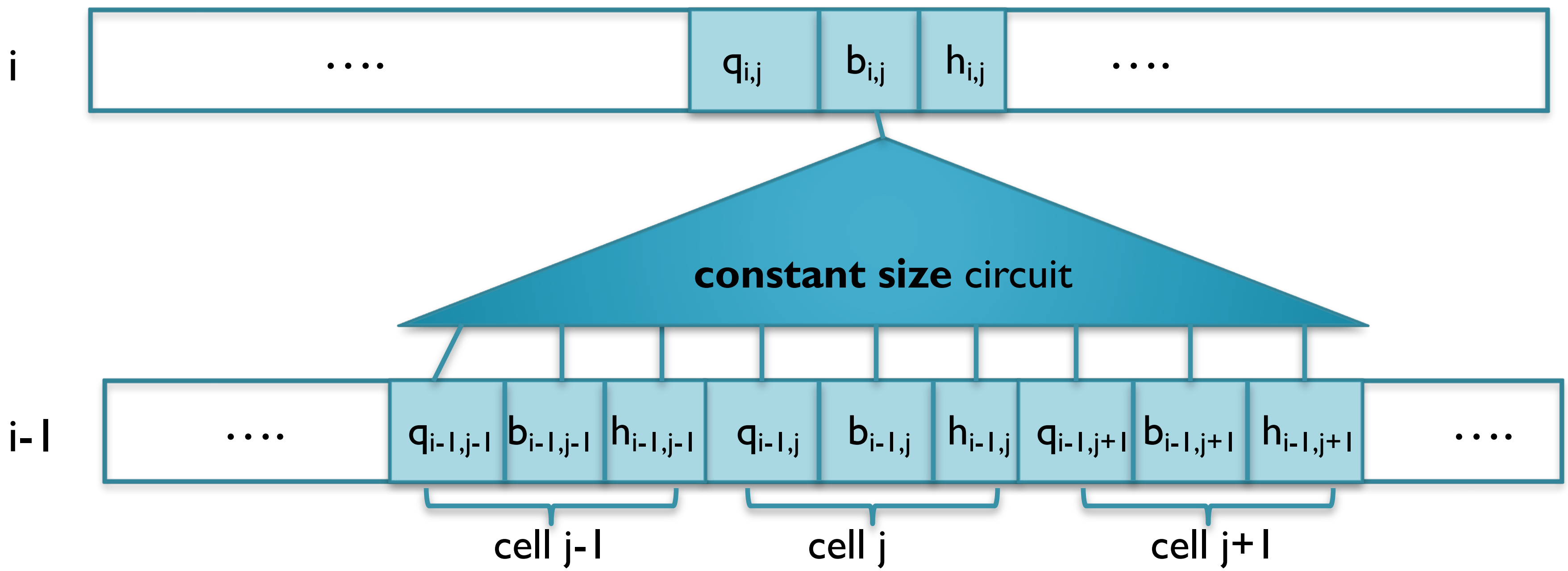
Main theorem: Step 1

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

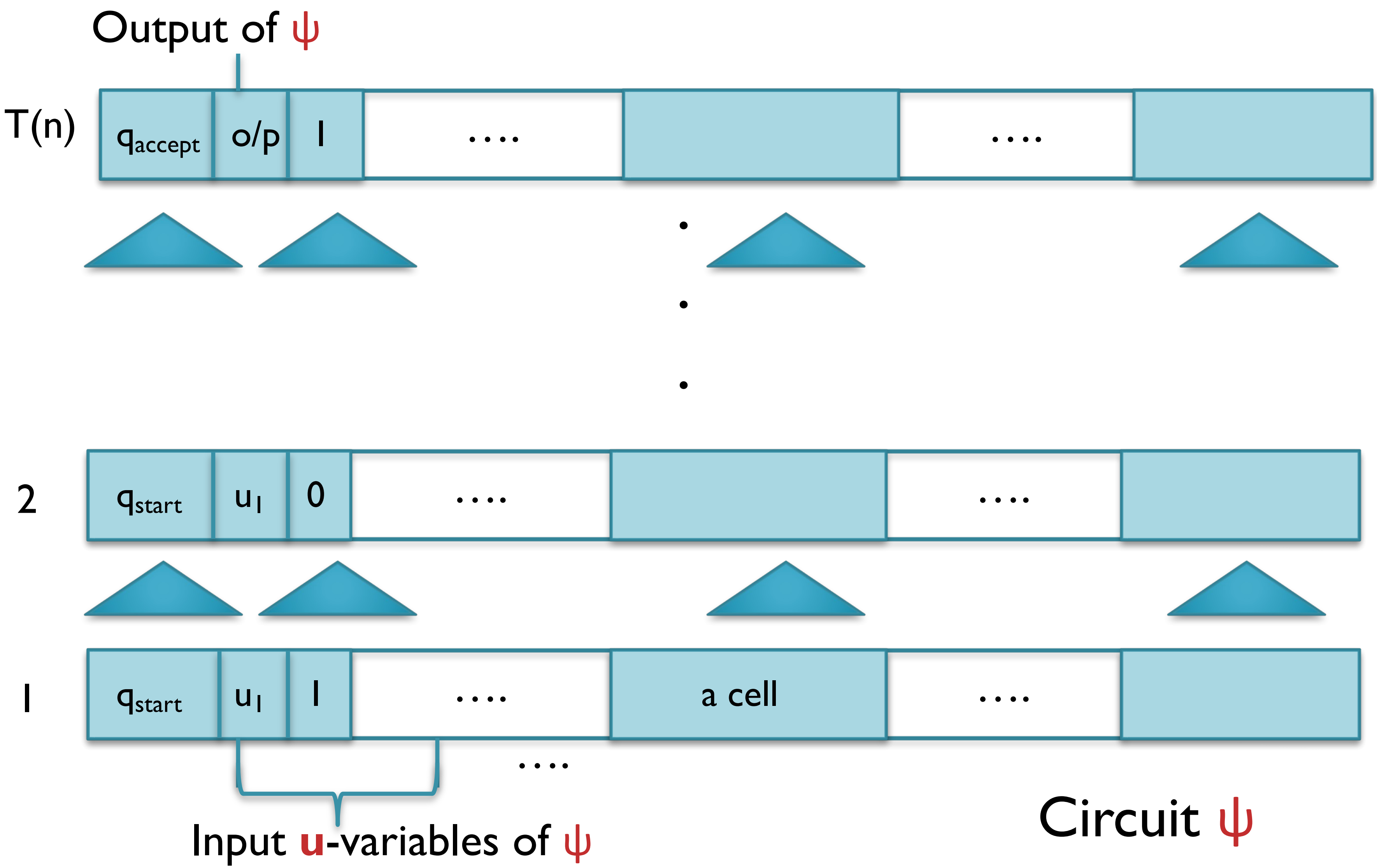


Main theorem: Step 1

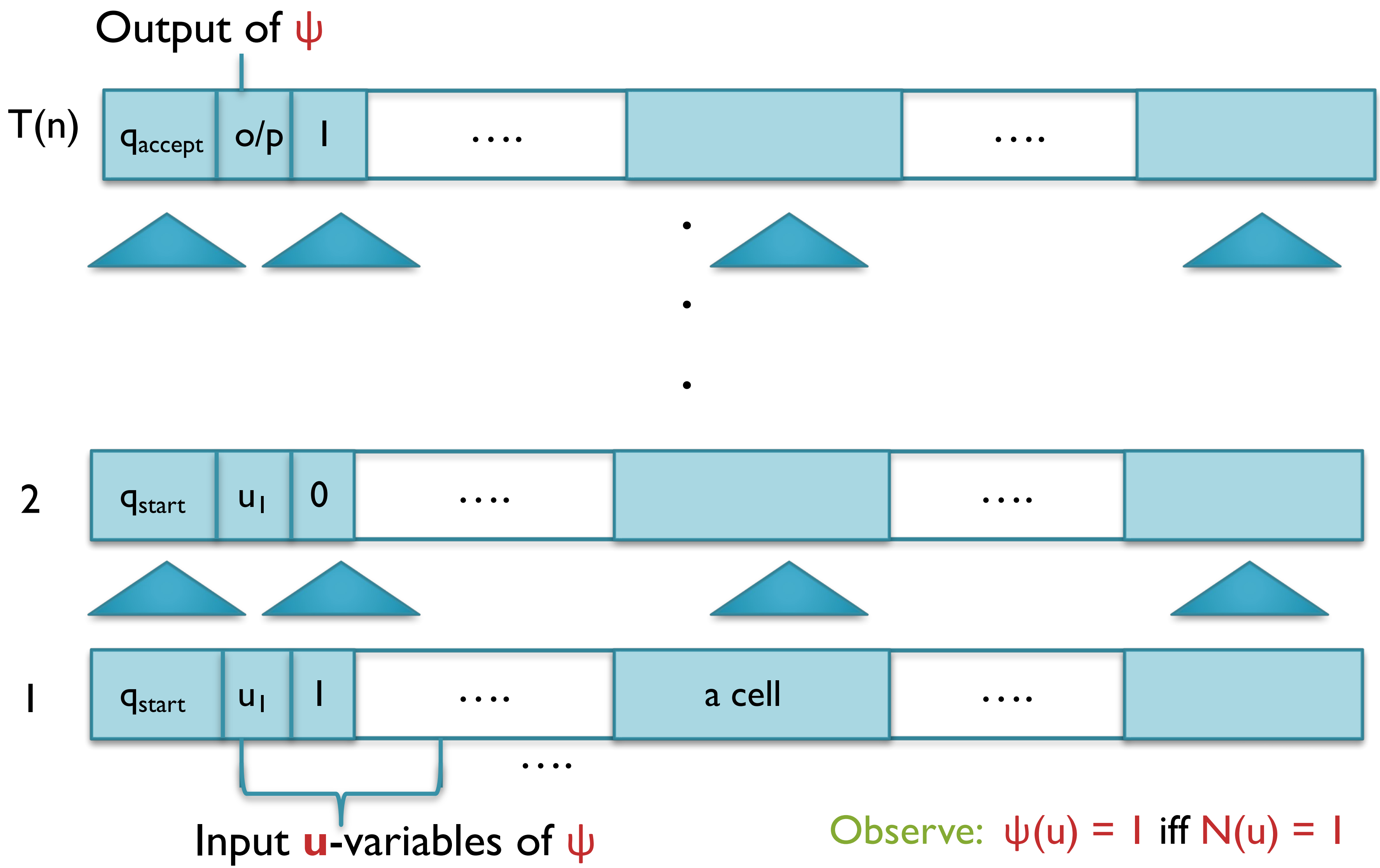
- **Locality of computation:** The bits in $h_{i,j}$, $b_{i,j}$ and $q_{i,j}$ depend ***only on*** the bits in
  - $h_{i-1,j-1}$, $b_{i-1,j-1}$, $q_{i-1,j-1}$,
  - $h_{i-1,j}$, $b_{i-1,j}$, $q_{i-1,j}$,
  - $h_{i-1,j+1}$, $b_{i-1,j+1}$, $q_{i-1,j+1}$



Main theorem: Step 1



Main theorem: Step 1



Observe: $\psi(u) = 1$ iff $N(u) = 1$

Recall Steps 1 and 2

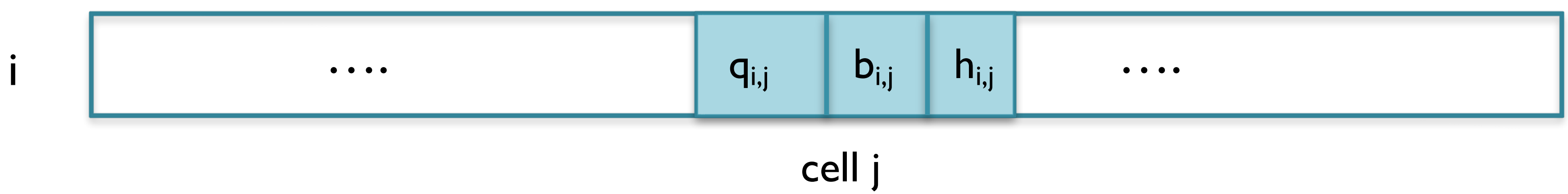
- Step 1. Let N be a deterministic TM that runs in time T(n) on every input u of length n, and outputs 0/1. Then,
    1. There's a Boolean circuit ψ of size *poly*(T(n)) such that ψ(u) = 1 if and only if N(u) = 1.
    2. ψ is computable in time *poly(T(n))* from N, T & n.

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

Main theorem: Step 2

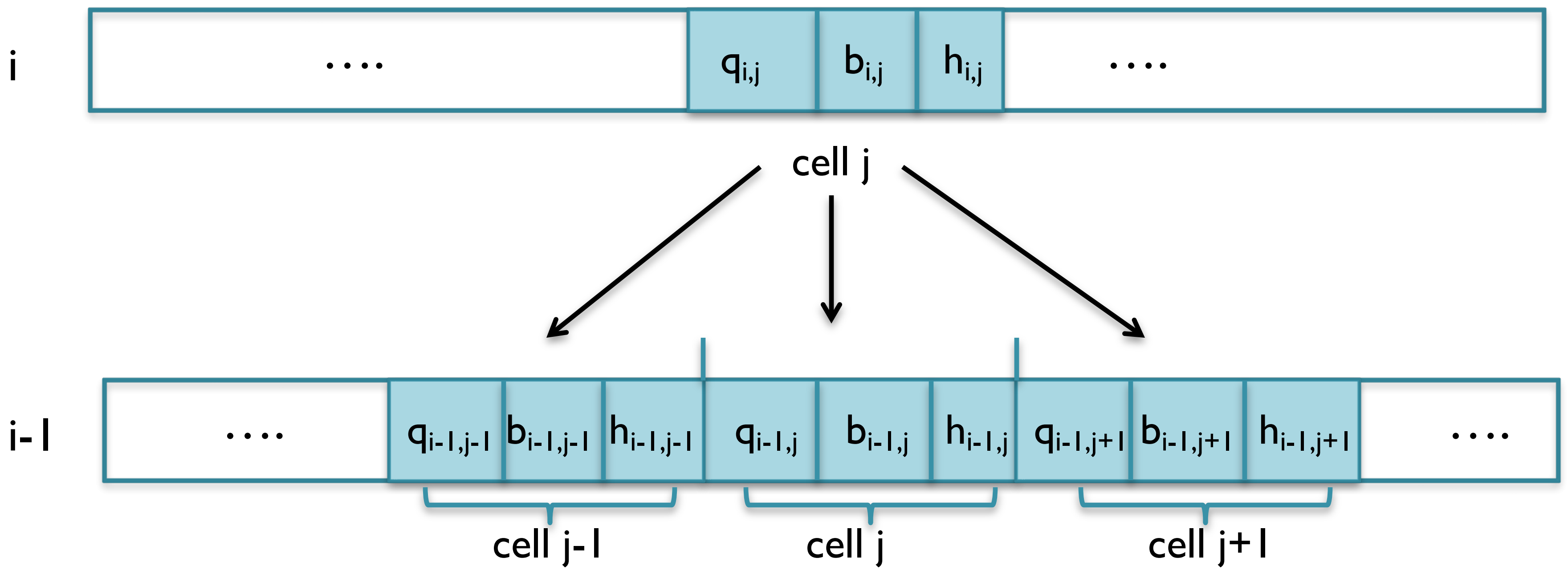
- Think of $h_{i,j}$, $b_{i,j}$ and the bits of $q_{i,j}$ as formal Boolean variables.

auxiliary variables



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

- 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.

$x = y$ iff $(x \wedge y) \vee (\neg x \wedge \neg y) = 1.$

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 <u>constant size</u> CNF Φ_{ij} . Also, Φ_{ij} is easily computable from δ.

Main theorem: Step 2

k-th bit of q_{ij} where $1 \leq k \leq \log |Q|$

- Similarly,

  $q_{ijk} = C_{ijk}(h_{i-1,j-1}, b_{i-1,j-1}, q_{i-1,j-1}, h_{i-1,j}, b_{i-1,j}, q_{i-1,j}, h_{i-1,j+1}, b_{i-1,j+1}, q_{i-1,j+1})$

  = 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 $h_{i,j}$, $b_{i,j}$ and $q_{i,j,k}$. for all i,j,k. $|\lambda|$ 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 $h_{i,j}$, $b_{i,j}$ and $q_{i,j,k}$. for all i,j,k. $|\lambda|$ is $O(T(n)^2)$.
- Define $\varphi = \lambda \wedge b_{T(n),1}$.

Main theorem: Step 2

- Observe: An assignment to u and the auxiliary variables satisfies λ if and only if it "captures" the computation of N on the assigned input u for T(n) steps.

Main theorem: Step 2

- Observe: An assignment to u and the auxiliary variables satisfies λ if and only if it "captures" the computation of N on the assigned input u for T(n) steps.

- Hence, an assignment to u and the auxiliary variables satisfies φ if and only if N(u) = 1, i.e., for every u,

$$\varphi(u, \text{"}\textit{auxiliary variables}\text{"}) \in SAT \iff N(u) = 1.$$

Recall the Main Theorem

- 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.

Main theorem: Comments

- φ is a CNF of size $O(T(n)^2)$ and is also computable from N, T and n in $O(T(n)^2)$ time.

- Remark 1. With some more effort, size φ can be brought down to $O(T(n). \log T(n))$.

- Remark 2. The reduction from x to φ_x 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 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 a called a k-CNF if every clause has at most k literals.

  e.g. a 2-CNF $\varphi = (x_1 \lor x_2) \land (x_3 \lor \neg x_2)$

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

3SAT is NP-complete

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

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

- 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.

3SAT is NP-complete

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

  e.g. a 2-CNF $\varphi = (x_1 \lor x_2) \land (x_3 \lor \neg x_2)$

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

- Theorem. *(Cook-Levin)* 3-SAT is NP-complete.