## Problem

Show that a circuit family with depth O(log n) is also a polynomial size circuit family.

## Step-by-step solution

**Step 1** of 2

A circuit family of depth $O(\log n)$ can be obtained by taking an equivalent polynomial size family of formulas. **"To convert a formula $\mu$ with $h$ leaves to a similar circuit of depth $O(\log h)$" is sufficient to show the above statement.** Here, it may be assumed that fan-in value of all the nodes is 2 and not gates are pushed to the leaves.

- Now, the proof of $h \geq 4$ (can be done using the induction hypothesis) that is a formula $\mu$ with $h$ leaves is similar to a formula $\mu'$ with a maximum depth of $C \log_2 h$ .where the value of the constant C will be further determined.
- If $h \leq 4$, suppose $\mu' = \mu$. Otherwise apply the concept of tree which says that every tree with $m \geq 2$ leaves has a sub-tree with between $m/3$ and $2m/3$. By applying this concept, the tree structure of $\mu$ acquire a sub-formula $\beta$ with between $h/3$ and $2h/3$.

Comment

**Step 2** of 2

**Suppose** $\hat{\mu}(y)$ be $\mu$ with the sub-formula $\mu$ is replace by a new variable $y$. Thus, $\mu$ is similar to $\hat{\mu}(\beta)$ and $\mu$ is equivalent to $\mu_1$ that is given by:

$$\mu_1 = \left(\beta \wedge \hat{\mu}(1)\right) \vee \left(\neg\beta \wedge \hat{\mu}(0)\right)$$

Here, $\hat{\mu}(1)$ and $\hat{\mu}(0)$ each contains maximum $2h/3$ leaves which is variable.

- Finally, suppose $\mu$ and $\mu'$ with the equivalent of the sub formulas $\beta$, $\hat{\mu}(1)$ and $\hat{\mu}(0)$ interchanged by similar small depth formula given by the induction hypothesis.
- **Thus, the depth of $\mu'$ is maximum** $C \log_2\left((2/3)h\right) + 3$. This is maximum of $C \log_2 h$ provided $C \geq 3 / \log_2\left(3/2\right)$.
- Thus, from the above explanation it can be said that "**A circuit family of depth $O(\log n)$ can be obtained by taking an equivalent polynomial size family of formulas**".

Comment

## Problem

Show that 12 is not pseudoprime because it fails some Fermat test.

## Step-by-step solution

**Step 1** of 2

A number is said to be pseudoprime if the number passes Fermat test. This means the number should be prime related to all the numbers that are less than the given number.

Fermat primality test is a test used for checking whether the number is prime or not. A number is said to be prime if the number satisfies following condition:

$$p^{n-1} \equiv 1 \pmod{n}$$

where $\mathrm{p} \geq 1$ and $\mathrm{n} > \mathrm{p}$

The equation given above can be written as:

$$p^{n-1} - 1 = nk$$

where *k* is an positive integer.

Comment

**Step 2** of 2

Fermat test for 12 is given below:

$$4^{12-1} - 1 = 12k$$
$$4^{11} - 1 = 12k$$

$$4194304 - 1 = 12k$$
$$4194303 = 12k$$

When 4194303 is divided with 12k, this will provide remainder 3. So, 12 fails Fermat test. So, 12 is not pseudoprime.

Comment

## Problem

Prove that if A $\leq_L$ B and B is in NC, then A is in NC.

## Step-by-step solution

**Step 1** of 2

**If** $A \leq_L B$ **and** $B$ **is in** $NC$ **then it can be proved that** $A$ **is also in** $NC$. This can be achieved by using the fact of circuit evaluation. In other word, this can be achieved by showing that" **the problem of circuit evaluation is** $P$ **complete".**

For a circuit $C$ and input string $w$, the value of $C$ on $w$ can be written as $C(w)$. **Suppose**

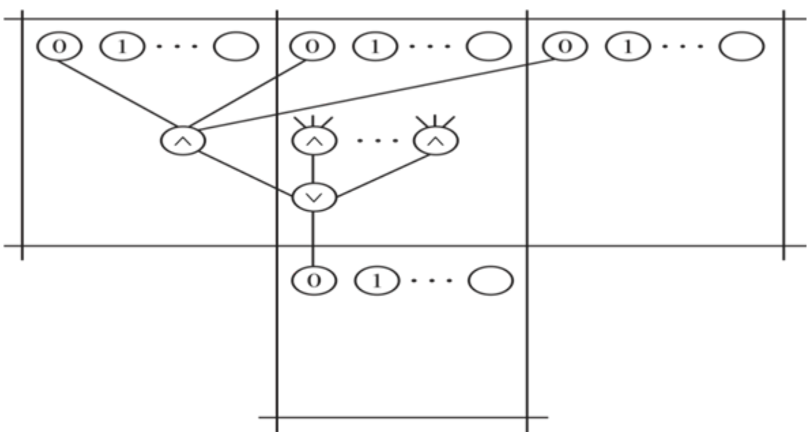
$CIRCUIT-VALUE = \{\langle C, x\rangle \mid C \text{ is a Boolean circuit and } C(x) = 1\}$

Comment

**Step 2** of 2

Consider the given theorem, which says that "**suppose** $t: M \rightarrow M$ **be a function, where** $t(m) \geq m$. **If** $W \in TIME(t(m))$**, then the complexity of the circuit** $A$ **is given by** $O(t^2(m))$

• Now, consider the figure which is given below:



• The above figure shows the construction of the theorem, which is discussed above. It shows the way of **reduction of a language** $W$ **(which is in** $P$**)** to $CIRCUIT-VALUE$.

• On input $w$, the production of a circuit takes place by the reduction. The process reduction simulates the Turing machine for $W$ in polynomial time. The $w$ itself can be taken as an input to the circuit.

• A $\log \text{space}$ is used to carried out the reduction because **the circuit produced by it contains a repetitive and a simple structure.**

Hence, **it shows that "** $CIRCUIT-VALUE$ **is** $P$**-complete" and the circuit produced by it has a repetitive structure.** Therefore it can be said that "**If** $A \leq_L B$ **and** $B$ **is in** $NC$ **then** $A$ **is also in** $NC$."

Comment

## Problem

Show that the parity function with n inputs can be computed by a branching program that has O(n) nodes.

## Step-by-step solution

**Step 1** of 1

**A branching program** is defined as "**a directed acyclic graph** where labels of all the nodes are maintained by the variables, except for two output nodes labeled 1 or 0. Here, all the nodes **whose labels are maintained by the variables** are called **query nodes**. Every query nodes consists of two outgoing edges: one is labeled 1 and another one is labeled 0.

• So, from the definition of branching program as defined above "the n-input parity function can be computed by a branching program that consists $O(n)$ nodes.

• It can be achieved by **building a binary tree of gates that compute XOR function**, **where XOR function is used as equivalent to the parity function. The implementation of each XOR gate can be done by using two AND's, two NOT's and one OR gates.**

• As the implementation of XOR function consists a binary tree of different gates. Here, the output node of binary tree is labeled 1 and 0, which acts a branching program.

**Therefore**, from the above explanation it can be said that "**the n-input parity function can be computed by a branching program that consists** $O(n)$ **nodes**".

Comment

## Problem

Show that the majority function with n inputs can be computed by a branching program that has O($n^2$) nodes.

## Step-by-step solution

### **Step 1** of 2

**A branching program**is defined as "**a directed acyclic graph** where the variables are used to label all the nodes except only two output nodes which is labeled $0$ and $1$". The **query nodes** are defined as all the nodes whish are labeled by the variables. All the query nodes consists two outgoing edges, labeled as 0 and 1. Both output nodes doesn't consists outgoing edges.

Now, consider about the **function majority** $\left(majority_n : \{0,1\}^n \rightarrow \{0,1\}\right)$, that is defined as:

$$majority_n\left(x_1, x_2, ..., x_n\right) = 0 \ if \sum x_i < n/2;$$
$$=1 \ if \sum x_i \geq n/2.$$

The computation of the **majority function** can be done by using **a branching program**, which consist $O\left(n^2\right)$nodes.

Comment

### **Step 2** of 2

Now, suppose the number of inputs taken is $n$. **A bubble-sort can be implemented as a circuit**. It is used to compare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as $x_1, x_2$and the outputs can be called as $y_1, y_2$. A sub-circuit can be written which accomplishes this as $\mathbf{y_1 = OR\left(x_1, x_2\right)}$and $\mathbf{y_2 = AND\left(x_1, x_2\right)}$ . This will be act as a part of **branching program**. This circuit contains a size of two.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the $n$input, $n$-output sub-circuit that passes through all the inputs taken as $< k \ and \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k \ and \geq k+1st$ input can be used to generate $kth$ and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, ..., n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of $n$passes. Therefore, this gives a size $\mathbf{n(n-1)*2 = O\left(n^2\right)}$.

**Here, AND gates and OR gates are used to construct the branching program. Therefore, it can be said that "a branching program with $O\left(n^2\right)$ nodes can be used to compute the majority function with $n$ inputs.**

Comment

## Problem

Show that any function with n inputs can be computed by a branching program that has $O(2^n)$ nodes.

## Step-by-step solution

**Step 1** of 4

**A program** is defined as a branching program if "**a directed graph, which also shows an acyclic** property, where labels of all the given nodes are maintained by the variables. These variables will not be used for two output nodes which are labeled 1 or 0. Here, all the nodes **whose labels are maintained by the variables** are also known as **query nodes**. Every query nodes consists of two edges which is outgoing from itself. In which one output node is labeled as 1 and another one is labeled as 0.

Comment

**Step 2** of 4

Now, consider the **majority function** $\left(majority_n : \{0,1\}^n \rightarrow \{0,1\}\right)$, which is defined as:

$$\begin{aligned} majority_n\left(x_1, x_2, \ldots, x_n\right) &= 0 \ \text{if} \sum x_i < n/2; \\ &= 1 \ \text{if} \sum x_i \geq n/2. \end{aligned}$$

As it is known "the computation of the **majority function** can be done by using **a branching program** of $O\left(2^n\right)$ nodes"

Comment

**Step 3** of 4

Again, consider the function **parity**; it is known that "**a branching program with** $O(n)$ **gates may be used to compute the n-input parity function".**

• It can be achieved by **building a binary tree of gates which is used to calculate function XOR**, **where the function XOR is used as equivalent to the parity function. The implementation of each XOR gate can be done by using two AND's, two NOT's and one OR gates.**

• It is known that each **AND's, OR gates takes two nodes as an input and produces a single output w**hich again considered as an input of another gate because a binary tree of gates is considered**.**

• Finally, the total number of nodes, which are used in computation, is an order of $2^n$ or $O\left(2^n\right)$.

Comment

**Step 4** of 4

As discussed above, the **majority** and the **function parity, which** takes $n$-inputs, can be obtained by a branching program, which consist $O\left(2^n\right)$ nodes. Therefore, **as the definition of branching program says (which is explained above), it can be said that any function, that consists** $n$**-inputs, can be computed by using a branching program which consist** $O\left(2^n\right)$ **nodes.**

Comments (1)

## Problem

Show that BPP $\subseteq$ PSPACE.

## Step-by-step solution

**Step 1** of 1

Consider $M$ be a probabilistic $TM$ which runs on polynomial time.

Therefore, $M \in BPP$. $M$ can be modified so that $M$ makes exactly $n^k$ coin tosses in each branch of its computation, for any constant $k$.There are total of $2^{(n^k)}$computation paths. Hence, problem which is determining probability that $M$ accepts its input is reduces to counting how many branches, $B$, are accepting and comparing this number with $P = \left(\frac{3}{4}\right).2^{(n^k)}$.

If $B \geq P$, then accept; otherwise reject. Now the given deterministic task can be performed in the polynomial space by generating all possible paths sequentially following *M*'s program but recycling the space used by the previous path.

Hence, the problem of $BPP$ is converted into $PSPACE$. Therefore, $BPP \subseteq PSPACE$.

Comment

## Problem

Let A be a regular language over {0,1}. Show that A has size–depth complexity (O(n),O(log n)).

## Step-by-step solution

**Step 1** of 1

Suppose a **language A** is defined in $\{0,1\}$ that consist every strings with an odd number of one's. The parity function computation can be used to test membership in A. The standard AND, OR and NOT operations can be used **to implement the two parity gate** $a \oplus b$ as $(a \wedge \neg b) \vee (\neg a \wedge b)$.

• Suppose $a_1, a_2, \ldots, a_n$ be taken as the input to the circuit. There are many ways define a circuit with $O(n)$ size. One way to get a circuit for parity function is to construct gates $g_i$ whereby $g_1 = a_1$ and $g_i = a_i \oplus g_{i-1}$ for $i \le n$. This construction uses $O(n)$ size and depth.

• Another way to do this, by **building a binary tree of gates** that computes the XOR function, where the XOR function is the same as the parity function and then implements **each XOR gate with two NOTs, two ANDs and one OR gates**. This construction uses $O(n)$ and $O(\log n)$ depth.

This construction is a significant improvement because it **uses exponentially less parallel time** than does the preceding construction. **Thus, the size-depth complexity of A is** $(O(n), O(\log n))$.

Comment

## Problem

A ***Boolean formula*** is a Boolean circuit wherein every gate has only one output wire. The same input variable may appear in multiple places of a Boolean formula. Prove that a language has a polynomial size family of formulas iff it is in $NC^1$. Ignore uniformity considerations.

## Step-by-step solution

**Step 1** of 2

**A Boolean formula** is defined as a Boolean circuit which consist only a single output wire for every input gate. The Boolean formula may consists the same input variable at many places. Here, it can be shown that **a polynomial size family of formulas can be used to compute all the languages in $\mathbf{NC^1}$**.

A normal induction hypothesis on $d$ is used to show that "a formula, whose size is less than $O(2^d)$ is similar to every Boolean circuit of depth $d$". For each step of the induction, the circuit's output gate is considered in such a way that the maximum fan-in value acquired is 2. The induction hypothesis can also be applied to each input gate.

• The $n$th circuit $C_n$ has depth $O(\log n)$ in an $\mathbf{NC^1}$ circuit family. Therefore, the equivalent formula has size $2^{O(\log n)} = n^{O(1)}$ that is polynomial in size.

Comment

**Step 2** of 2

**To prove its converse**, first it need to proof that every tree with $h \geq 2$ leaves consist a sub-tree with between $h/3$ and $2h/3$ leaves.

• Suppose a binary tree is denoted by $B$ with $h \geq 2$ leaves. Beginning at the parent (root) of $B$, and traverse towards the child's (leaves), always taking a sub-tree with minimum half of the number of the existing sub-tree.

• Finally stop this iteration when a sub-tree $B^{'}$ is reached which consists at most $2h/3$ leaves. Then, $B^{'}$ will contain minimum of $h/3$ leaves as the previous sub-tree consists more than $2h/3$ leaves. Thus, the desired sub-tree is $B^{'}$.

**Thus**, from the above explanation, it can be said that **a polynomial size family of formulas can be used to compute all the languages in $\mathbf{NC^1}$**.

.

Comment

## Problem

A k-***head pushdown automaton*** (k-PDA) is a deterministic pushdown automaton with k read-only, two-way input heads and a read/write stack. Define the class $PDA_k$ = {A| A is recognized by a k-PDA}. Show that $\mathrm{P} = \bigcup_k \mathrm{PDA}_k.$ (Hint:Recall that P equals alternating log space.)

## Step-by-step solution

**Step 1** of 3

A deterministic pushed down automation, which consists $k$ read-only, a read write stack **and two ways input heads**, can be defined as a $k$**-head pushed-down automation** ( $k-\mathrm{PDA}$). Consider the **class** $\mathrm{PDA}_k$, which is defined as:

$\mathrm{PDA}_k = \{A \mid A\, is\, recognized\, by\, a\, k-\mathrm{PDA}\}$

Now, by using the above given facts, $\mathrm{P}=\bigcup_k \mathrm{PDA}_k$ has to be proved. It can be achieved by using, $ASPACE(s(n)) = TIME\left(2^{O(s(n))}\right)$, which shows that $\mathrm{P}$ is alternating log-space.

Comment

**Step 2** of 3

Now, a machine $M$, with deterministic time $2^{O(s(n))}$, is constructed. It is used to simulate an alternating space $O(s(n))$ machine $S$. If an input $q$ is given to the simulator $M$, a graph is constructed by the simulator for the computation of $S$ on $q$.

- The nodes are configured for $S$ on $q$ which use maximum $ls(n)$ space, where $l$ is defined as the constant factor approximation for $S$.
- Edges are drawn from a configuration to those other configuration which can be generated in a single move of $S$.
- After the construction of graph, $M$ iteratively scans it and marks those configurations which are accepting.
- Initially the acceptance configuration is marked. After that all the universal branching is marked is all of its children marked as an accepting state. Machine $S$ continued marking and scanning until no additional nodes are marked on scan.

Comment

**Step 3** of 3

As it is given that, $s(n) \geq \log n$, the configuration's number of $S$ on $q$ is $2^{O(s(n))}$. Hence the configuration graph's size is given by $2^{O(s(n))}$ and its construction may be done in $2^{O(s(n))}$ time.

- It takes roughly the same time to **scan the graph once**. Here, the **number of scans** is **equal to the maximum number of nodes in the graph**. Hence, the total time used is $2^{O(s(n))}$.
- Now, from the above discussion it can be said that $ASPACE(s(n)) = TIME\left(2^{O(s(n))}\right)$. It also shows, $P$ **is alternating log-space.**

**Hence, for** $\mathrm{PDA}_k = \{A \mid A\, is\, recognized\, by\, a\, k-\mathrm{PDA}\}$, it can be said that $\mathrm{P}=\bigcup_k \mathrm{PDA}_k$.

Comment

## Problem

LetM be a probabilistic polynomial time Turing machine, and let C be a language where for some fixed $0 < \epsilon_1 < \epsilon_2 < 1$,

**a.** $w \notin C$ implies $\Pr[M \text{ accepts } w] \le \epsilon_1$, and

**b.** $w \in C$ implies $\Pr[M \text{ accepts } w] \ge \epsilon_2$.

Show that $C \in$ BPP. (Hint: Use the result of Lemma 10.5.)

## Step-by-step solution

**Step 1** of 1

Given $M$ be probabilistic Turing Machine and $C$ be a language where for some fixed $0 < \varepsilon_1 < \varepsilon_2 < 1$,

$1.\ w \notin C$ implies $\Pr[M \text{ accepts } w] \le \varepsilon_1$

$2.\ w \in C$ implies $\Pr[M \text{ accepts } w] \le \varepsilon_2$

It is required to show $C \in BPP$.

• Between any two distinct real numbers $\varepsilon_1 < \varepsilon_2$ there exists another real number that lies strictly between them. Thus to choose $c$ such that $\varepsilon_1 < c < \varepsilon_2$.

• Consider another machine $S$ which repeatedly runs $M$. Now $S$ accepts if the proportion of $M's$ acceptance is greater or equal to $c$, and $S$ rejects if the proportion of $M's$ acceptance is less than $c$. Now to show $S$ decides in $BPP$.

• Consider the variable $S_k$ be the total number of acceptances by machine $M$ after $k$ runs on input $w$. Hence, for $w \in C$, $S_k$ is the sum of $k$ $0-1$ random variables with common mean $\mu_2 > \varepsilon_2$, and for $w \notin C$, $S_k$ is sum of $k$ $0-1$ random variable with common mean $\mu_1 > \varepsilon_1$. The error probabilities can then be expressed as follows:

$$1.\text{For } w \in C, \Pr[S \text{ rejects } w] = \Pr\left[\frac{S_k}{k} < c\right] \le \Pr\left[\left|\frac{S_k}{k} - \mu_2\right| > \mu_2 - c\right]$$

$$2.\text{For } w \in C, \Pr[S \text{ accepts } w] = \Pr\left[\frac{S_k}{k} \ge c\right] \le \Pr\left[\left|\frac{S_k}{k} - \mu_1\right| \ge c - \mu_1\right]$$

By the weak law of large numbers (or various other bounds from probability theory), there exist $k$ that will make those probabilities on the right as small as desired, and in particular, there exist $k$ that will make them both strictly less than $\frac{1}{2}$.

• By using "Amplification lemma" this shows $C \in BPP$.

Comment

## Problem

Show that if P = NP, then P = PH.

## Step-by-step solution

**Step 1** of 1

It is required to show that if $P = NP,$ then $P = PH$.

• Firstly, if $P = NP$, then because $P$ is closed under complement, thus $P = C_o NP$. Written as, $P = \sum_1 P = \prod_1 P$.

• Now using induction that if $P = \sum_i P = \prod_i P,$ then $P = \sum_{i+1} P = \prod_{i+1} P$.

1. Assume $\sum_{i+1} P$ machine $M$ ,that consists of a run of the existential branching, then existential branching etc.

2. Assume the computation sub-tree path whose root are first universal step along path. For each such type of sub-tree, $M$ is performing a $\prod_i$ computation. By hypothesis, $\prod_i P = P$.

3. Thus for the forming of new machine $S$ each of computation sub-trees can be replaced by deterministic (non- branching) polynomial time of computation.

4. If assume $a(n)$ be the number of maximum steps which are taken by other machine before the start of universal machine, $P(n)$ be the maximum steps which are taken by any deterministic which were substituted for $\prod_i$ computations in $P$ machines, therefore $S$ is covered by $a(n) + p(a(n))$. Remember that the $p(a(n))$ term is composition of the functions, because $P$ sub procedures with inputs are computing which may be a longer than $n$ (but it must be equal or smaller than $a(n)$, since only $a(n)$ steps are executed on the time the sub procedures are used).

5. Since $a$ and $P$ both are polynomials, Therefore, $S$ is in $NP$. By hypothesis $P = NP$, so $S$ is in $P$ as well.

6. A similar type of argument may be used to reduce a $\prod_{i+1} P$ machine to $P = C_o NP$ machine , hence, putting it in $P$ as well, and completing collapse of hierarchy.

Comment

## Problem

Show that if PH = PSPACE, then the polynomial time hierarchy has only finitely many distinct levels.

## Step-by-step solution

**Step 1** of 2

The hierarchy in polynomial-time exists between deterministically accepted languages of classes $\mathrm{P}$ **in polynomial time** and deterministically or non-deterministically accepted languages of class $\mathrm{PSPACE}$ **in polynomial space**. There exists a relativization, which allowed minimum three levels of the hierarchy. The number of distinct levels, which are used to determine "low" and "high", in the hierarchy of polynomial-time are, depends upon the perishing of the $\mathrm{NP}$ class. Now consider the facts of "low" and "high", which is explained below:

• If there exist some $\mathrm{i}$ for which $\sum_{\mathrm{i}}^{\mathrm{P}}(Y)\subseteq\sum_{i}^{\mathrm{P}}$ then a set $\mathrm{E}$ in $\mathrm{NP}$ is known as "low" and if there exists $\sum_{\mathrm{i+1}}^{\mathrm{P}}\subseteq\sum_{i}^{\mathrm{P}}(z)$ for some $\mathrm{i}$, then it is known as "high".

Comment

**Step 2** of 2

Now, suppose $\mathrm{PH}$ **is defined as the union of the various classes of polynomial time hierarchy**. From the explanation of "low" and "high" (as it is defined above) , it can be shown that "**the hierarchy collapses are the only way of simultaneously existence of high and low for a set of PH.**

• There are two principle results exists. First one is based on the fact "either all or none (that is, every sparse set in $\mathrm{PH}$ is high or no one is low. Second one is that "every set of sparse will be extended high or no one sets will be extended high.

• Simply, it can be said that "**the hierarchy collapses in polynomial-time are** the only way for simultaneous existence of high and low behavior".

• A disjoint set can be obtained by combining high sets and low sets. The reason behind it is "the existence of immeasurably many levels extended by the hierarchy in polynomial time and in $\mathrm{NP}$ there exist some sets which show neither low nor high.

• **Therefore, the hierarchy in polynomial-size can be extended to only distinct finitely level if a sparse set in** $\mathrm{NP}$**.** Hence, it can be said that **if** $\mathrm{PH{=}PSPACE}$**, then the polynomial time hierarchy consists levels which are distinct and finite**.

Comment

## Problem

Recall that $NP^{SAT}$ is the class of languages that are decided by nondeterministic polynomial time Turing machines with an oracle for the satisfiability problem. Show that $NP^{SAT} = \Sigma_2 P$.

## Step-by-step solution

**Step 1** of 1

It is required to show $NP^{SAT} = \Sigma_2 P$.

1. Firstly, to show that $\Sigma_2 P \subseteq NP^{SAT}$.

• Suppose a language L is decided by $\Sigma_2$ alternating Turing machine $M$. Then that Turing machine performs some number of existential branching, followed by some number of universal branching.

• Consider the sub-trees of the computation path whose roots are the first universal step along the path. For each such sub-tree, $M$ is (by definition) performing a $\Pi_2$ computation. Thus, it is deciding a language in $Co\text{-}NP$ (Note: it may possibly be a different language for each sub-tree, but it does not matter).

• Now consider a nondeterministic machine $S$ that behaves just as $M$ does, but replaces each of the computation sub-trees discussed above with a deterministic computation by a machine in $P^{SAT}$ (this is feasible because $Co-NP \subseteq P^{SAT}$). Note that this machine only contains one run of existential branches, each extended with a deterministic computation that uses an $SAT$ oracle and runs in polynomial time.

• If assume $a(n)$ be the maximum number of steps taken by the alternating machine before the universal branches start, and $p(n)$ be the maximum number of steps taken by any of the $P^{SAT}$ machines which have substituted for the universal branching, then the running time of $S$ is bounded by $a(n)+p(a(n))$. Note that the $p(a(n))$ term is a composition of functions, because the $P^{SAT}$ sub procedures are computing with inputs that may be longer than $n$ (but must be smaller than or equal to $a(n)$, since only $a(n)$ steps have been executed at the time the sub-procedures are used).

• Since $a$ and $p$ are both polynomials, so is their composition. Therefore, $S$ is in $NP^{SAT}$.

2. Next, to show that $NP^{SAT} = \Sigma_2 P$.

• Suppose a language $L$ is decided by a $NP$ machine $M$ with an $SAT$ oracle.

• Consider the following modification of $M$. At each step of $M's$ computation that depends on the result of an oracle query, replace the step with a nondeterministic “split" that guesses the answer to the query. Each branch of the “split" also writes down the queried formula and answer obtained. Thus, at every “leaf" node of the computation tree (whenever the computation terminates), the tape contains a record of all queries and answers which lead to that leaf. Finally, in order to preserve the proper accepting behavior of $M$, replace each accepting “leaf" of $M$ with a computation that checks that all the oracle.

• This last computation can be done purely with a run of existential branches followed by a run of universal branches. First, all the positively answered queries are checked by running an NP machine that decides SAT. Second, all of the negatively answered queries are checked by running a $Co\text{-}NP$ machine that decides $UNSAT$.

• Finally, we check that the machine we have produced is indeed in $\Sigma_2 P$. Clearly, the steps have been ordered so that all of the existential branches precede all of the universal branches. Now it is just required to check that the maximum branch length is polynomial with respect to the input. There are at most a polynomial number of oracle queries and replacing each of them with a branch that writes the formula and query result only lengthens the branch by a polynomial amount. Thus the total branch lengthening caused by this replacement is only a polynomial times a polynomial. Since the “checking" step at the end is performed by a polynomial number of applications of $NP$ and $Co\text{-}NP$ machines (each run on a formula that is at most polynomial larger than the original input), the total time taken for this “checking" step is polynomial as well. Therefore, the machine is in $\Sigma_2 P$.

Comment

## Problem

Prove Fermat's little theorem, which is given in Theorem 10.6. (Hint: Consider the sequence $a_1, a_2, \ldots$. What must happen, and how?)

**THEOREM 10.6**

If $p$ is prime and $a \in \mathcal{Z}_p^+$, then $a^{p-1} \equiv 1 \pmod p$.

## Step-by-step solution

**Step 1** of 1

**Statement:** if $p$ be a prime and $a \in Z^+{}_p$ then $a^{p-1} \equiv 1(\bmod p)$. Here $Z^+{}_p$ is defined as $Z^+{}_p = \{1,\ldots,p-1\}$ and $(p,a)$ is co-prime.

**Proof:** consider the following first $p-1$ positive multiple of $a$.

$a, 2a, 3a, \ldots, (p-1)a$.

• As the **little Fermat's theorem** states " $(p,a)$ is co-prime (that is, $p$ is not exactly divisible by $a$)". Suppose $xa$ and $ya$ are taken in such a way that, the modulo $p$ of $xa$ and $ya$ are equal.

• Now, it can be said that $\mathbf{x} = \mathbf{s}(\bmod p)$. So the $p-1$ multiples by $a$ above are non-zero and distinct; that is, they must be congruent to $a, 2a, 3a, \ldots, (p-1)a$ in the same order. Now, multiply **all these congruence together** and which gives:

$a, 2a, 3a, \ldots, (p-1)a = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1)(\bmod p)$

$a^{p-1}(p-1)! = (p-1)!(\bmod p)$

• Now, dividing each side by $(p-1)!$ in the above equality

$\mathbf{a^{p-1} \equiv 1(\bmod p)}$.

**It is also known as a little Fermat's theorem and sometimes it can also be represented as**

$a^p = a(\bmod p)$.

Comment

## Problem

Prove that for any integer p > 1, if p isn't pseudoprime, then p fails the Fermat test for at least half of all numbers in $Z^+_p$

## Step-by-step solution

**Step 1** of 1

It sufficient to prove that elements set in $Z_p^+$ that pass the Fermat test forms multiplicative subgroup of $Z_p^+$. Since the subgroup order divides the group order, if subgroup is a strict subgroup, it must contain at most half of elements of group.

To show that the set is a subgroup, it is required to show that it is nonempty and closed under the inverses and multiplication.

• First, the set is nonempty, since $1^{p-1} \equiv 1 \text{ mod p}$.

• If $a^{p-1} \equiv 1 \bmod p$, and $b^{p-1} \equiv 1 \bmod p$, then $(ab)^{p-1} \equiv a^{p-1}b^{p-1} \equiv 1 \bmod p$, which shows closure under multiplication.

• If $a^{p-1} \equiv 1 \bmod p$, then multiplying both sides of the equation by the $(a^{-1})^{p-1}$ shows that $1 \equiv (a^{-1})^{p-1} \bmod p$. Thus, the set is closed under inverses.

Hence, on the other side if $p$ is not pseudo prime then $p$ fails Fermat Test for at least half of number.

Comment

## Problem

Prove that if A is a language in L, a family of branching programs ($B_1$,$B_2$, . . .) exists wherein each $B_n$ accepts exactly the strings in A of length n and is bounded in size by a polynomial in n.

## Step-by-step solution

**Step 1** of 2

**A branching program** is defined as "**a directed acyclic graph** where the variables are used to label all the nodes except only two output nodes which is labeled $0$ and $1$". The **query nodes** are defined as all the nodes which are labeled by the variables. All the query nodes consists two outgoing edges, labeled as 0 and 1. Both output nodes doesn't consists outgoing edges.

Comment

**Step 2** of 2

Consider a language $A$ which takes an **input length** of $n$. A set of **branching programs** is taken in such a way that each branching program accepts exactly the strings in $A$ of length.

• Now, the **Merge-sort** can be implemented as a circuit in which the input length of language $A$ has taken as the nodes of the branching program.

• It is used to compare two bits after recursively dividing the given inputs in to half. The total time taken here (to divide the inputs into equal halves iteratively) is $\log n$.

• Consider the inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1$. Now, **the action of the merge-sort algorithm** can be mimicked on an array. It can be implemented one step at position to be the $n$ input, $n/2$**-output sub-circuit**.

• Now, **a pass** can be implemented as the serial concatenation of steps, which has a size $n \log n$. Therefore, this gives a size $n * \log n = O(n \log n)$.

• Therefore, it can be said that "a language with an input length of $n$ can be computed in $O(n \log n)$ size circuits by using branching program.

**Hence from the above explanation it can be said that the language $A$ is in logarithmic space. In other words, the language the language $A$ is in $L$.**

Comment

## Problem

Prove that if A is a regular language, a family of branching programs ($B_1$,$B_2$, . . .) exists wherein each $B_n$ accepts exactly the strings in A of length n and is bounded in size by a constant times n.

## Step-by-step solution

**Step 1** of 4

A formal language, which can be expressed using a regular expression, is called as a **regular language**. In other words, it can be defined as a language which is recognized by a finite automation. All the languages which are finite are regular.

• Now consider a **regular language** $A$**,** then a family of branching program $(B_1, B_2, ...)$ in which a string, of length $n$ in language $A$, is accepted by each $B_n$ and is **bounded by a fixed time** $n$ **in size.** It can be achieved by a way which is given below.

Comment

**Step 2** of 4

Now consider a branching program**. A branching program** is known as "**a directed acyclic graph** where labels of all the nodes are maintained by the variables, except for two output nodes which are labeled as 1 or 0.

• All the nodes **whose labels are maintained by the variables** are called **query nodes**.

• Every query nodes consists of two outgoing edges: one is labeled 1 and another one is labeled 0. Both output nodes doesn't consists outgoing edges.

Comment

**Step 3** of 4

So, from the definition of **branching program** and **regular language** $A$ as defined above "the n-input function or a finite regular language can be computed by a branching program that consist a constant $O(n)$ size.

• A **bubble-sort** can be implemented as a circuit $n$. A set of **branching programs** is taken in such a way that each branching program accepts exactly the strings in $A$ of length $n$.

• It is used to compare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1, y_2$.

• A sub-circuit can be written which accomplishes this as $y_1 = OR(x_1, x_2)$ and $y_2 = AND(x_1, x_2)$. **This circuit contains a size of two**.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the $n$ input, $n$-output sub-circuit that passes through all the inputs taken as $< k \; and \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k \; and \geq k+1st$ input can be used to generate $kth$ and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, ..., n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of one passes. Therefore, this gives a size $1\mathbf{(n-1)*2 = O(n)}$.

Comment

**Step 4** of 4

Hence, from the above explanation it can be said that "**a family of branching program** $(B_1, B_2, ...)$ **in which a string, of length** $n$ **in language** $A$**, is accepted by each** $B_n$ **and is bounded by a fixed time** $n$ **in size if the language** $A$ **is regular**.

## Problem

Show that if NP $\subseteq$ BPP, then NP = RP.

## Step-by-step solution

**Step 1** of 2

As $RP \in NP$, it is sufficient to show $NP \subseteq RP$. It can be prove by showing that if $NP \subseteq BPP$, then $SAT \in RP$. Consider a formula $\phi$ with $n$ variables $x_1, x_2, \ldots x_n$ be the input. $\phi$ is satisfied iff $\exists$ truth assignment for $x_1, x_2, \ldots x_n$ so that $\phi(x_1, x_2 \ldots x_n) = 1$.

• Assume $A$ be a $BPP$ algorithm with error probability at most $2^{-k}$ for $SAT$, where $k = |\phi|$ is the length of formula $\phi$. Such $A$ exist because of the assumption that $SAT \in BPP$.

• First run $A$ on $\phi$. If $A$ rejects, then reject, otherwise try to satisfy assignment for $\phi$ on variable at a time.

• Initialize $x_1$ to $0$ and then call $A$ to determine that if the formula resulting is satisfiable: if $A$ returns "accept" then permanently set $x_1$ to $0$; otherwise set $x_1$ to $1$. Then proceed with $x_2$ similarly.

• If manage to construct a satisfying assignment at end then verify this assignment for $\phi$. If $\phi(x_1, \ldots, x_n) = 1$, then accept; otherwise reject.

• Here is the analysis. If $\phi$ is unsatisfiable, then always reject either because $A$ rejects in the process or do not arrive at a satisfying assignment at the end.

Comment

**Step 2** of 2

On the other hand, suppose $\phi$ is satisfiable. Proceed to show that it accept with probability at least $\frac{1}{2}$.

• Invoke $A$ a total of $n+1$ times. If $\phi$ is satisfiable and $A$ returns "accept" each time only for an assignment for variable $x_i$ which is part of a satisfying assignment, then end up with a satisfying assignment.

• Now show that the probability that at least one of the $n+1$ invocations returns "reject" for an assignment for variable $x_i$ which is part of a satisfying assignment is at most $\frac{1}{2}$.

• The probability that an invocation of $A$ returns does so is at most $2-k$. So probability that encounter it is at most $(n+1).2^{-k}$, which is at most $\frac{1}{2}$ because $n+1 \leq k$.

Since both the algorithm $A$ and the construction of satisfying assignment run in polynomial time, the whole procedure clearly runs in polynomial time. Hence, $SAT \in RP$ and therefore $NP = RP$.

Comment

**Problem**

Define a ZPP-***machine*** to be a probabilistic Turing machine that is permitted three types of output on each of its branches: *accept*, *reject*, and *?*. A ZPP-machine M decides a language A if M outputs the correct answer on every input string w $(accept \text{ if } w \in A \text{ and } reject \text{ if } w \notin A)$ with probability at least $\frac{2}{3}$, and $M$ never outputs the wrong answer. On every input, M may output *?* with probability at most 1/3 . Furthermore, the average running time over all branches of M on w must be bounded by a polynomial in the length of w. Show that RP ∩ coRP = ZPP, where ZPP is the collection of languages that are recognized by ZPP-machines.

**Step-by-step solution**

**Step 1** of 2

- $RP \bigcap co-RP \subseteq ZPP$. Consider $L \in RP \bigcap co-RP$.

Then assume $A$ be an $RP$ algorithm for $L$ and assume $B$ be an $co-RP$ algorithm for $L$.

• Consider $w$ be the input. One step of our $ZPP$ algorithm will be the following:

1. Run $A$ on $w$.

2. Then run $B$ on $w$ .

3. If $A$ accepts, accept.

4. If $B$ reject, reject.

Note that this step takes polynomial time since each $A$ and $B$ take polynomial time.

5. If $A$ rejects and $B$ accepts, repeat.

• First, show correctness.

Note that if $w \notin L$ then $A$ always rejects, so if $A$ accepts $w \in L$.

Similarly, if $B$ reject $w \notin L$. Thus, when output an answer, it is we always correct (which is better than probability $\frac{2}{3}$). Note that our algorithm never outputs $?$[1].

• Then, we prove that the running time is polynomial in expectation. Note that since each step has polynomial running time, it suffices to show the number of steps is polynomial in expectation (in fact, we will show it is constant in expectation). Note that if $w \in L$, then $A$ will accept with the probability at least $\frac{1}{2}$ (by definition of $RP$). Similarly, for $w \notin L$ and $B$ reject. Thus, every step succeeds with the probability of at least $\frac{1}{2}$ .

• Assume $X$ is a random variable denoting how many steps to take. Then let $p_k = \Pr[X = k]$ denote the probability by taking exactly $k$ steps $k-1$. Note that $p_k$ is the probability that we fail on the first $k-1$ steps and then succeed, so

$$E|X| = \sum_{k=1}^{\infty} \frac{k}{2^{k-1}} = 4$$

And this is done[2].

Comment

**Step 2** of 2

[1] In fact, $ZPP$ actually stands for zero-error probabilistic polynomial-time. This comes fact that it is equivalent to define $ZPP$ algorithms as never outputting a wrong answer.

[2] General Formula can be drive to do last sum $(\text{for} |x| < 1)$:

$$\sum_{k=1}^{\infty} kx^{k-1} = \sum_{k=1}^{\infty}\left[\frac{d}{dx}x^k\right]$$
$$= \frac{d}{dx}\left[\sum_{k=1}^{\infty} x^k\right]$$
$$= \frac{d}{dx}\frac{1}{(1-x)}$$
$$= \frac{1}{(1-x)^2}$$

- $ZPP \subseteq RP$ (It can be proved that $ZPP \subseteq co-RP$ analogously) consider $L$ be in $ZPP$, and $A$ be an $ZPP$ algorithm for $L$, and consider $p(n)$ be the expected running time of $A$ on input of length $n$. Then build the following $RP$ algorithm:

• On input $w$ of length $n$,

1. Run $A$ on $w$ for time $6p(n)$.

2. If $A$ accepts or rejects, output the answer $A$.

3. Otherwise (If $A$ does not terminate, or outputs $?$), reject.

• Note that since $A$ has expected polynomial running time $p(n)$ is a polynomial, so $6p(n)$ is also polynomial. It suffices to prove correctness. First, if $w \notin L$, then $A$ will either reject, print $?$ or not terminate. In all of these cases, reject, so it is correct.

• On the other hand, assume $w \in L$. Again, three things can happen: $A$ accepts, $A$ prints $?$, or $A$ does not terminate in time. We are correct in the first two cases, so it suffices to show the probability of the latter two are at most $\frac{1}{2}$. The probability that $A$ outputs $?$ is at most $\frac{1}{3}$.

• The probability that $A$ does not terminate in 6p(n) time is at most $\frac{1}{3}$ (by Markov's inequality). Thus, if $w \in L$, the probability that $A$ is wrong is at most $\frac{1}{2}$, thus this algorithm is an $RP$ algorithm for $L$.

Comment

## Problem

Let $EQ_{BP}$ = { $\langle B_1, B_2 \rangle$ | $B_1$ and $B_2$ are equivalent branching programs}. Show that $EQ_{BP}$ is coNP-complete.

## Step-by-step solution

**Step 1** of 2

Consider $EQ_{BP} = \{\{B_1, B_2\} \mid B_1 \text{ } and \text{ } B_2 \text{ both are equivalent branching program}\}$

Branching programs $B_1$ and $B_2$ can be described by acyclic graph rejects or may accept input strings $s_1, s_2, \ldots, s_n$. For turing machines these problems are un-decidable but these problems are $coNP$ complete for circuits.

Comment

**Step 2** of 2

Consider the problem $EQ_{BP}$ that is $B_1 \text{ } and \text{ } B_2$ is restricted to read-once programs. By using the equivalence with $coRP$ for testing the equivalence, and by reduction from $co\text{-}3SAT$ it will be $coNP$.

Polynomial can be determined by following way:

- Assign the vertex in programs $EQ_{BP}$ for branching, from root to final states.
- Label all incoming edges, now vertex polynomials will be sum of polynomials of edge which are incoming.
- Polynomial which is associated with final state $1$ will be branching program polynomial.

As the branching program is read-once, and have power not more than one. Hence polynomial cannot be more than degree of $n$. Hence, $EQ_{BP}$ must be $coNP$.

Comment

## Problem

Let BPL be the collection of languages that are decided by probabilistic log space Turing machines with error probability 1/3 . Prove that BPL $\subseteq$ P.

## Step-by-step solution

**Step 1** of 1

Suppose $\mathbf{BPL}$ be the collection of languages which are judged by **probabilistic log space** TM (Turing Machine) with an error probability of $1/3$. Now, suppose $L$ be a $\mathbf{BPL}$ language and **the machine** $M$ **is required** as the definition of $\mathbf{BPL}$ says.

- On input $x$ of length $n$, suppose the number of configuration of $M(.,x)$ is defined as $C$. A $C \times C$ matrix is constructed in such a way that $P[c_1, c_2] = 1/3$ if $c_2$ is reachable from $c_1$ in a single step, and $P[c_1, c_2] = 0$ otherwise.
- For all $t$, $P^t[c_1, c_2]$ is defined as the **probability of approaching configuration** $c_2$ **from configuration** $c_1$ in $t$ number of steps. Here, $P^t$ is defined as the matrix obtained by multiplying $P$ with itself $t$ times.
- The accepting probability of $M(.,x)$ can be **computed by** computing all powers of $P$ till the running time of $M(.,x)$ and **decide** if $x \in L$.
- The exact calculation can be performed at this time: **each probability is an integer multiple of** $1/3^{p(n)}$. **So, the polynomial number of digits can be used to represent it**.

Hence, it can be said that $\mathbf{BPL} \subseteq \mathbf{P}$.

Comment

## Problem

Let *CNF*$_H$ = { $\langle \phi \rangle | \ \phi$ is a satisfiable cnf-formula where each clause contains any number of literals, but at most one negated literal}. Problem 7.25 asked you to show that *CNF*$_H$ ? P. Now give a log-space reduction from *CIRCUIT VALUE* to *CNF*$_H$ to conclude that *CNF*$_H$ is P-complete.

## Step-by-step solution

**Step 1** of 4

Consider the following $CNF_H$ statement:

$CNF_H = \{<\varnothing>|\varnothing$ is a satisfiable cnf-formula, where every clause consists any number of literals, but is consists maximum one negated literals $\}$

It is known that $\mathbf{CNF_H \in P}$.

Comment

**Step 2** of 4

Now, consider the **circuit evaluation** $CIRCUIT-VALUE$. For a circuit $C$ and input string $w$, the value of $C$ on $w$ can be written as $C(w)$. Then, $CIRCUIT-VALUE$ is given by

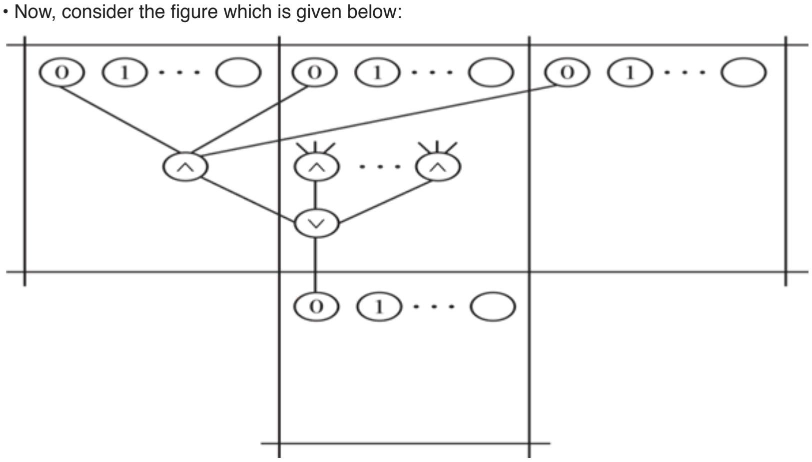
$CIRCUIT-VALUE = \{\langle C, x\rangle \mid C \, is\, a\, Boolean\, circuit\, and\, C(x)=1\}$

Comment

**Step 3** of 4

Consider the given theorem, which says that "**suppose** $t: M \rightarrow M$ **be a function, where** $t(m) \geq m$. **If** $W \in TIME(t(m))$**, then the complexity of the circuit** $A$ **is given by** $O(t^2(m))$

• Now, consider the figure which is given below:




• The above figure shows the construction of the theorem, which is discussed above. It shows the way of **reduction of a language** $W$ **(which is in** $P$**)** to $CIRCUIT-VALUE$.

• On input $w$, the production of a circuit takes place by the reduction. The process reduction simulates the Turing machine for $W$ in polynomial time. The $w$ itself can be taken as an input to the circuit.

• **A log-space is used to carry out the reduction because** the circuit produced by it contains a repetitive and a simple structure. **It shows that "** $CIRCUIT-VALUE$ **is** $P$**-complete.**

Comment

**Step 4** of 4

The above explanation **gives a log-space reduction from** $CIRCUIT-VALUE$ to $CNF_H$. Hence, from the above discussion it can be concluded that " $\mathbf{CNF_H\ is\ P\text{-}complete}$ ".

Comment