# Theory of Computation

Final Examination on January 14, 2021
Fall Semester, 2020

**Problem 1 (20 points)** Consider any CNF $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where literals in a clause are distinct. Suppose $m \geq 2^{\epsilon n}$. Show that for any $\epsilon > 0$, there is a polynomial-time algorithm which solves SAT for CNFs on $n$ variables where each clause has at least $\epsilon n$ distinct literals.

**Proof:** If a clause contains a variable and its negation, remove that clause as it is trivially satisfiable. Then the length of $\phi$ is at least $2^{\epsilon n}$. An exhaustive search of all $2^n$ truth assignments becomes polynomial in the input length. ■

**Problem 2 (20 points)** Prove that if NP $\subseteq$ ZPP, then NP $\subseteq$ BPP.

**Proof:** Assume NP $\subseteq$ ZPP. Pick any NP-complete language $L$. We only need to show that $L \in$ BPP. There exists a Las Vegas algorithm A that decides $L$ in expected polynomial time, say $p(n)$. By Markov's inequality, the probability that the running time of A exceeds $3\,p(n)$ is at most $1/3$. Run A for $3\,p(n)$ steps to determine with probability at least $1 - 1/3 = 2/3$ whether the input belongs in $L$. We therefore obtain a polynomial-time algorithm for $L$ which errs with probability at most $1/3$ on each input. Hence $L$ is in BPP. ■

**Problem 3 (20 points)** Present interactive proofs of QR (Quadratic Residuacity) and QNR (Quadratic Nonresiduacity).

**Proof:** In an interactive proof of QR, the prover sends the square root of the input $x \in Z_n^*$ to the verifier.

- If $x$ is a quadratic residue, the verifier can verify the square root is valid in polynomial time.
- If $x$ is a quadratic nonresidue, the verifier will reject the incorrect square root.

In an interactive proof of QNR, the prover sends $n$'s factorization $p$ and $q$ to the verifier.

- The verifier checks $pq = n$ and applies Lemma 82 (see p. 687 of the slides) to check if $x$ is a quadratic nonresidue. Both can be done in polynomial time. This protocol can also solve QR.

■

**Problem 4 (20 points)** Define IP$^*$ as IP except that the prover now runs in deterministic polynomial space instead of exponential time. Show that IP$^*$ $\subseteq$ PSPACE. (You cannot use the known fact IP = PSPACE.)

**Proof:** Let $L \in \mathbf{IP}^*$, $(P, V)$ be an interactive proof system, $V$ be a probabilistic polynomial-time verifier, $P$ be a polynomial-space prover, $k$ be some positive integer, $n$ be the length of the input $x$, $r \in \{0,1\}^{n^k}$ be the random bits used by $V$, and $a$ be the number of $r$'s for which $V$ accepts. To prove the claim, we will describe a deterministic algorithm that runs in polynomial space and decides the same language $L$. This algorithm simulates $(P, V)$ by trying all possible random bits that $V$ uses and calculating the ratio of the number of ACCEPT to the number of all possible random bits, while recycling the space. The detail is described below. On any input $x$, $M$ computes $a$ as follows:

1: $a = 0$;
2: **for all** $r \in \{0,1\}^{n^k}$ **do**
3: Simulate $(P, V)$ deterministically;
4: **if** $V$ outputs ACCEPT **then**
5: $a = a + 1$;
6: **end if**
7: Free the tape and start over;
8: **end for**
9: **return** $a$;

Step 3 takes polynomial space. If $a \,/\, 2^{n^k} \geq 2/3$, then $M$ accepts $x$; otherwise, $M$ rejects $x$. This algorithm performs in polynomial space. So $M$ decides $L$ in polynomial space. ■

**Problem 5 (20 points)** Suppose that there are $n$ jobs to be assigned to $m < n$ identical machines. Let $t_i$ be the running time for job $i \in \{1, 2, \ldots, n\}$, $A[\,i\,] = j$ be an assignment for job $i$ on machine $j \in \{1, 2, \ldots, m\}$, and $T[\,j\,] = \sum_{A[\,i\,]=j} t_i$ be the total running time for machine $j$. For convenience, assume that $t_1 \leq t_2 \leq \cdots \leq t_n$. The makespan of $A$ is the maximum time that any machine is busy, given by

$$\text{makespan}(A) = \max_j T[\,j\,].$$

The problem LoadBalance is to compute the minimal makespan of $A$. It is known that the decision version of LoadBalance is NP-hard. Consider the following algorithm for LoadBalance:

1: **for** $i \leftarrow 1$ to $n$ **do**
2: $A[\,i\,] \leftarrow \emptyset$;
3: **end for**
4: **for** $i \leftarrow 1$ to $m$ **do**
5: $T[\,i\,] \leftarrow 0$;

```
6: end for
7: for i ← 1 to n do
8:     min ← 1;
9:     for j ← 2 to m do
10:        if T[j] < T[min] then
11:            min ← j;
12:        end if
13:    end for
14:    A[i] ← min;
15:    T[min] ← T[min] + t_i;
16: end for
17: return max_i T[i];
```

Show that this greedy algorithm for LoadBalance is a $\frac{1}{3}$-approximation algorithm, meaning that it returns a solution that is at most 3/2 times the optimum.

**Proof:** Let OPT be the optimal makespan. It is clear that $\text{OPT} \geq \frac{1}{m}\sum_{i=1}^{n} t_i$. By the pigeonhole principle, at least one machine in the optimal scheduling must get two of the first $m+1$ jobs. Each of these jobs is at least as big as $t_{m+1}$. Hence $\text{OPT} \geq 2\,t_{m+1}$. Suppose that machine $j^*$ has the largest total running time, and let $i^*$ be the last job assigned to machine $j^*$. Since $T[j^*] - t_{i^*} \leq T[j]$ for all $j \in \{1, 2, \ldots, m\}$, $T[j^*] - t_{i^*}$ is less than or equal to the average running time over all machines. Thus,

$$T[j^*] = (T[j^*] - t_{i^*}) + t_{i^*} \leq \frac{1}{m}\sum_{j=1}^{m} T[j] + t_{m+1} = \frac{1}{m}\sum_{i=1}^{n} t_i + t_{m+1} \leq \frac{3}{2} \times \text{OPT}.$$

■