Guidelines:

Please submit your assignment to the TA/lecturer at the beginning of the appropriate lecture. **Solutions should be typed**, drawings can be done by hand.

You are encouraged to work in groups of up to three people, however we ask that you dedicate enough time to think about each problem by yourself before consulting others. Moreover, **you must write up each problem solution by yourself without assistance**. You are asked to identify your collaborators on problem sets. If you did not work with anyone, please write “Collaborators: none”.

You may consult the course notes and the text (Sipser). When this problem set references a section or a problem, we mean the corresponding section or problem of the textbook Introduction to the Theory of Computation (3rd Edition) by Michael Sipser.

The assignment questions may be similar to questions on problem sets from past offerings of this course or courses at other universities. Using any preexisting solutions from these or any other sources is strictly prohibited.

Each of the Questions 1-6 is worth 10 points, please attempt all problems. Please provide full proofs, unless stated otherwise.

**Question 1.**

Consider the following problem of hash tables: Let $n$ be a large number and let $S \subseteq \{0, 1\}^n$ be a set such that $|S| \ll 2^n$. We wish to store the elements of $S$ in a table of size $\approx |S|$, such that we can efficiently check whether a string $x \in \{0, 1\}^n$ is in $S$ or not. A common solution to this problem is using hash tables: Let $2^k$ denote the size of the table in which we want to store $S$. We choose a function $h: \{0, 1\}^n \to \{0, 1\}^k$. Then, for every $i \in \{0, 1\}^k$, we store in the $i^{th}$ cell of the table all the elements $x \in S$ such that $h(x) = i$. Whenever we are required to check whether a string $x$ is in $S$, we read the $h(x)^{th}$ cell of the table and check if it contains $x$. If each cell in the table contains only a few elements of $S$, then this solution is very efficient.

How do we make sure that each cell in the table does not contain too many elements? We need to choose $h$ such that there are not many collisions, i.e., there are not many pairs of elements $x, y \in S$ such that $h(x) = h(y)$. In order to achieve it, we choose a function $h$ that maps the elements of $\{0, 1\}^n$ to $\{0, 1\}^k$ in a “random-like” manner. If $h$ is “sufficiently
random” and if \( k \) is sufficiently large comparing to \( S \), then we indeed expect that there won’t be many collisions. The reason is that, if we would have chosen \( h(x) \) and \( h(y) \) completely at random, then the probability that \( h(x) = h(y) \) would have been \( 2^{-k} \), which is small. The problem is to choose a function \( h \) that has such a random property.

One possible solution for choosing such a function \( h \) is to choose \( h \) to be a random function from \( \{0,1\}^n \) to \( \{0,1\}^k \). The problem with this solution is that in order to compute \( h \) later, we will need to store in the memory a table that says for every \( x \in \{0,1\}^n \) what is \( h(x) \). Storing such a table will require \( k \cdot 2^n \) bits of memory, which is too expensive for many applications.

Our problem can be now stated as follows: We need to find a function \( h \) that behaves like a random function, even though it is not really a random function. Furthermore, we want to be able to compute \( h \) without using too much memory. This problem is very common in theoretical computer science, and arises in many contexts rather than just in the context of hash tables - in fact, this is one of the major problems of the theory of derandomization.

In order to solve this problem, we first need to define what does it mean for a function \( h \) to “behave like a random function”. One useful definition is known that of pairwise independence, or 2-universal, hash functions. We note that pairwise independence is not a property of a single function, but rather a property of a family of functions. A family of functions is said to be pairwise independent if when we choose a random function from this family, then the distribution of the images of the function is “somewhat random”. The formal definition is as follows:

**Definition (Pairwise independent hash functions):** Let \( H_{n,k} \) be a collection of functions \( h : \{0,1\}^n \to \{0,1\}^k \). We say that \( H_{n,k} \) is pairwise independent if for every \( x, x' \in \{0,1\}^n \) with \( x \neq x' \) and for every \( y, y' \in \{0,1\}^k \),

\[
Pr_{h \leftarrow H_{n,k}}[h(x) = y \land h(x') = y'] = 2^{-2k}.
\]

Observe that a random function is indeed a pairwise independent function. More formally, the set of all functions \( r : \{0,1\}^n \to \{0,1\}^k \), denoted \( R_{n,k} \), is pairwise independent. In fact, the set \( R_{n,k} \) satisfies a stronger condition: For every \( \ell \in [2^n] = \{1, \ldots, 2^n\} \) distinct element \( x_1, \ldots, x_\ell \in \{0,1\}^n \), and for every \( y_1, y_2, \ldots, y_\ell \in \{0,1\}^k \),

\[
Pr_{r \leftarrow R_{n,k}}[r(x_1) = y_1 \land r(x_2) = y_2 \land \ldots \land r(x_\ell) = y_\ell] = 2^{-\ell k}.
\]

In this question we assume that \( k \leq n \).

1. (3 points) Prove the random subsum principle: Recall that for \( x, a \in \{0,1\}^n \) the inner product of \( x \) and \( a \) is given by

\[
\langle x, a \rangle = \sum_{i \in [n]} x_ia_i \quad (\text{mod } 2).
\]
Let \( x \in \{0,1\}^n \setminus \{0^n\} \). That is, \( x \) is a non-zero \( n \)-bit string. Prove that

\[
\Pr_{a \leftarrow \{0,1\}^n}[(x,a)] = 1/2.
\]

That is, prove that the inner product of \( x \) with a random string \( a \) is 1 with probability 1/2.

**Hint:** Show that whenever \( \langle x,a \rangle = 1 \), there is an \( a' \) related to \( a \) such that \( \langle x,a' \rangle = 0 \).

**Remark:** The name “the random subsum principle” stems from the fact that by taking an inner product with a random vector \( a \), we are summing a random subset of \( x \)'s coordinates.

2. (3 points) For a matrix \( A \in \{0,1\}^{k \times n} \) and a vector \( b \in \{0,1\}^k \), define the function \( h_{A,b}(x) = Ax + b \), where all additions and multiplications are performed modulo 2. Let

\[
H_{n,k} = \{ h_{A,b} : A \in \{0,1\}^{k \times n}, b \in \{0,1\}^k \}.
\]

Prove that the family of functions \( H_{n,k} \) is pairwise independent.

**Hint:** Use Part (1).

3. (3 points) Identify \( \{0,1\}^n \) with a finite field \( \mathbb{F} \) consisting of \( 2^n \) elements (e.g., \( \mathbb{F} = GF(2^n) \)). For elements \( a, b \in \mathbb{F} \), define the function \( g_{a,b} : \{0,1\}^n \rightarrow \{0,1\}^k \) as follows:

\[
g_{a,b}(x) = k \text{ rightmost bits of the binary representation of } f_{a,b}(x),
\]

\[
f_{a,b}(x) = a \cdot x + b,
\]

where addition and multiplication performed in \( \mathbb{F} \). Let

\[
G_{n,k} = \{ g_{a,b} : a, b \in \mathbb{F} \}.
\]

Prove that the family of functions \( G_{n,k} \) is pairwise independent.

**Hint:** Prove that the pair \( (a, b) \) is determined by \( x, x', y, y' \). Recall that \( a, b \) are selected uniformly at random and independently from \( \mathbb{F} \), thus the probability of every such pair is \( 2^{-2n} \).

4. (1 point) Recall that we would like to have a family of hash functions for which the representation of a function in the set is succinct. How many bits are required to represent a hash function from \( H_{n,k} \)? How many bits are required to represent a hash function from \( G_{n,k} \)? Which is more succinct?
Question 2.

Let \( L = \{w \# w : w \in \{0, 1\}^*\} \).

1. (5 points) Prove that every deterministic streaming algorithm for the language \( L \) requires memory of least \( \Omega(n) \).

2. (5 points) Show a randomized streaming algorithm for the language \( L \) that only requires \( O(\log n) \) bits and errs with probability at most \( 1/n \). Recall that the length of the stream is not known in advance.

   Hint: Let \( n, p \in \mathbb{N} \), where \( p \) is a prime. Encode \( w \in \{0, 1\}^n \) by
   \[
h_w(r) = w_1 \cdot r^1 + w_2 \cdot r^2 + \ldots + w_n \cdot r^n \pmod{p}.
   \]
   Recall that \( h_w \) has at most \( n \) roots.
   You may also use Chebyshev’s theorem (a.k.a Bertrand’s postulate) that states that for all positive integers \( n \), there is a prime \( p \) such that \( n \leq p \leq 2n \).

Question 3.

In class you saw a trivial streaming algorithm using \( O(n(\ell + \log n)) \leq O(n\ell) \) bits of memory for computing the most frequent element (MFE) in an \( n \)-element stream with elements from \( \Sigma = [2^\ell] = \{1, \ldots, 2^\ell\} \). But a streaming algorithm only makes one pass over the input. Can you do better with multiple passes? Between passes, it is acceptable to do arbitrary computations on the current memory (but not to read any other input). You may assume for this question that \( 2^\ell > n^2 \).

1. (3 points) Prove that with \( 2^\ell \) passes over the data, you can compute the most frequent element using only \( O(\ell) \) bits of memory.

2. (3 points) Prove that with \( p \) passes over the data (for \( p \in [2^\ell] = \{1, \ldots, 2^\ell\} \)), you can compute the most frequent element using only \( O\left(\frac{2^\ell}{p} \cdot \ell\right) \) bits of memory.

   Hint: Generalize your solution to Part (1).

3. (4 points) Prove that with \( p \) passes over the data (with \( p \) in the same range) you can compute the most frequent element using only \( O\left(\frac{n}{p} \cdot \ell\right) \) bits of memory.
Question 4.

Construct a Turing machine that performs unary multiplication. The machine should check that the input is of the form $1^n\#1^m\#$. If this is indeed the case, at the end of the machine’s execution, the machine should accept and the tape should have $1^n\#1^m\#1^{nm}$ written on it. If the input is of the wrong format, the machine should reject.

For example, if the input is $11\#11111\#$, then when the machine halts, $11\#11111\#1111111111$ should be written on the tape, as $2 \times 5 = 10$.

You do not have to provide a formal proof, just show the construction and explain the main ideas.

What is the running time of your Turing machine (as a function of $n$ and $m$)?

Question 5.

A Turing machine $M$ is called oblivious if for every input $w$ and for every number $i \in \mathbb{N}$, the position of $M$’s head after $i$ steps depends only on $|w|$ and $i$. That is, the head movements of $M$ depend only on the input size, and do not leak additional information about the specific input being processed.

Let $L \subseteq \{0,1\}^*$ be a language, and let $M$ be a Turing machine that decides if $x \in L$ in time $T(n)$, where $n = |x|$. Suppose that $T$ is time-constructible, i.e.,

Definition (Time-constructible functions): A function $T : \mathbb{N} \to \mathbb{N}$ is called time-constructible if:

1. $T(n) \geq n$ for all $n \in \mathbb{N}$.
2. There exists a Turing machine $M_T$ that on input $x$ outputs $1^{T(|x|)}$, where $1^{T(|x|)}$ is a string of $T(|x|)$ ones.

Show that there exists a constant $C > 0$ and an oblivious Turing machine $M'$ that decides $L$ in at most $C \cdot T(n)^2$ steps. Here, $C$ is a constant that depends only on the number of states and the alphabet size of $M$, but not on $n$. Where did you use the fact that $T$ is time-constructible?

You do not have to provide a formal proof, just show the construction and explain the main ideas.

Hint: Use several tapes and mark the location of the head.

Remark: With more effort, one can improve the running time of $M'$ to $C \cdot T(n) \log T(n)$. 
Question 6.

Prove that a language \( L \) is RE if and only if it can be expressed as
\[
L = \{ x : \text{there exists } y \text{ for which } \langle x, y \rangle \in R \},
\]
where \( R \) is a decidable language. In other words, you need to prove that every language of this form is RE, and that every RE language \( L \) has a related decidable language \( R_L \) that allows it to be described in the form above.

*Hint:* Read the Heading “Enumerators” in Chapter 3 (pages 180-181). You may use Theorem 3.21. Observe that if \( x \in L \), it should be printed by the enumerator in some computational step.

*Remark:* Recall that the algorithms recognizing an RE language \( L \) all had the following structure: Going over all possible “witnesses” \( y \) to the fact that \( x \in L \). If such a witness is found, output accept (otherwise, continue searching). This question shows that this not a coincidence, as all RE languages can be formulated this way.