1. What is the best, average and worst-case asymptotic time complexity for performing $n$ (correct) \texttt{remove()} operations on a map that initially contains $2^n$ entries and implemented with:

- Unordered list
- Direct address table
- Sorted search table
- Hash table with chaining
- Hash table with linear probing
- Hash table with double hashing

\textbf{Solution.}

For the following theorems, I call the operation \texttt{n-remove()} the action of performing \texttt{n remove()} operations on a map of size $2n$. Also, the theorems nested under a section assume the stated map implementation.

- Unordered list

\textbf{Theorem 1.} \textit{In the best case, n-remove()} takes $O(n)$ time.

\textit{Proof.} The best case is where the first item is always the key being removed from the unordered list. Therefore each of the $n$ operations takes $O(1)$ time for removal, totalling $O(n)$ time. \hfill \square

\textbf{Theorem 2.} \textit{In the average case, n-remove()} takes $O(n^2)$ time.

\textit{Proof.} In the average case, half of the list will be searched for the key to be removed. The actual removal is $O(1)$ time once we found the element. So each removal will on average take $O(n)$ meaning $n$ removals will take $O(n^2)$. \hfill \square

\textbf{Theorem 3.} \textit{In the worst case, n-remove()} takes $O(n^2)$ time.

\textit{Proof.} Similar to the proof of Theorem 2 except that in the worst case the entire list will need to be searched for a removal. So $n$ worst case removals will cost $O(n^2)$ time. \hfill \square

- Direct address table

\textbf{Theorem 4.} \textit{In the best, average, and worst case, n-remove()} takes $O(n)$ time.

\textit{Proof.} For a direct address table, finding a key (and subsequently) removing a key always takes $O(1)$ time. Thus, $n$ removes takes $O(1)$ time. \hfill \square
• Sorted search table

**Theorem 5.** In the best case, \texttt{n-remove()} takes \(O(n \log n)\) time.

*Proof.* The best case removal of an array would be a key at the end of the array of elements. So to find this key would take \(O(\log n)\) time. So, \(n\) removals in the best case, costs \(O(n \log n)\) time. \(\square\)

**Note:** \(O(n)\) is also acceptable for best case if you assume an optimized search routine that looks at the beginning or ending of the table as well as the middle.

**Theorem 6.** In the average case, \texttt{n-remove()} takes \(O(n^2)\) time.

*Proof.* On average, \(n\) elements must be shifted in an array of size \(2n\). Searching for an arbitrary element will take on average \(O(\log n)\) time. Clearly, the removal dominates the complexity and in total \(n\) removals will take \(O(n^2)\) time. \(\square\)

**Theorem 7.** In the worst case, \texttt{n-remove()} takes \(O(n^2)\) time.

*Proof.* Similar to the proofs of Theorem 5 and Theorem 6, except that the worst case is removing from the front of an array. So in the worst case, \(O(n)\) elements are shifted per removal costing \(O(n)\) per removal and \(O(n^2)\) for \(n\) removals. \(\square\)

• Hash table with chaining

**Theorem 8.** In the best case, \texttt{n-remove()} takes \(O(n)\) time.

*Proof.* In the best case, finding for a hash-table always takes \(O(1)\) time. Subsequently, this implies the element in question will be at the front of a chain, implying \(O(1)\) time removal. Thus the total is \(O(n)\) for \(n\) removals. \(\square\)

**Theorem 9.** In the average case, \texttt{n-remove()} takes \(O\left(\frac{n^2}{N}\right)\) time.

*Proof.* In the average case, we assume uniform hashing. So on average \(O(n/N)\) elements exist in each chain and consequently \(O(n/N)\) to remove from that chain. Thus, in total \(O\left(\frac{n^2}{N}\right)\) for \(n\) removals. \(\square\)

**Theorem 10.** In the worst case, \texttt{n-remove()} takes \(O(n^2)\) time.

*Proof.* In the worst case, all elements are in a single chain, and the key to be removed takes \(O(n)\) time to find. Thus, a single removal takes \(O(n)\) time and \(n\) removals take \(O(n^2)\) time. \(\square\)
• Hash table with linear probing

**Theorem 11.** In the best case, \( n\text{-}remove() \) takes \( O(n) \) time.

*Proof.* In the best case, there will be no collisions for any search, and every removal takes \( O(1) \) time, totaling \( O(n) \) for \( n \) removals. \( \square \)

**Theorem 12.** In the average case, \( n\text{-}remove() \) takes \( O\left(\frac{nN}{N-n}\right) \) time.

*Proof.* In the average case, the uniform hashing assumption holds. Therefore, we expect \( O\left(\frac{N}{N-n}\right) \) collisions for each find and the same complexity for one removal. So, in total \( n \) removals takes \( O\left(\frac{n^2}{N-n}\right) \) time. \( \square \)

**Note:** \( O\left(\frac{nN}{N-n}\right) = O\left(\frac{n}{1-\frac{n}{N}}\right) \). You may also say \( \alpha \) for the load factor.

**Theorem 13.** In the worst case, \( n\text{-}remove() \) takes \( O(n^2) \) time.

*Proof.* In the worst case, there will be \( O(n) \) collisions, i.e., all keys have the same hash value. Thus, one removal takes \( O(n) \) time and \( n \) removals takes \( O(n^2) \) time. \( \square \)

• Hash table with double hashing

**Theorem 14.** In the best case, \( n\text{-}remove() \) takes \( O(n) \) time.

*Proof.* Similar logic to the proof of Theorem 11. \( \square \)

**Theorem 15.** In the average case, \( n\text{-}remove() \) takes \( O\left(\frac{nN}{N-n}\right) \) time.

*Proof.* Similar logic to the proof of Theorem 12. \( \square \)

**Note:** \( O\left(\frac{nN}{N-n}\right) = O\left(\frac{n}{1-\frac{n}{N}}\right) \). You may also say \( \alpha \) for the load factor.

**Theorem 16.** In the worst case, \( n\text{-}remove() \) takes \( O(n^2) \) time.

*Proof.* Similar logic to the proof of Theorem 13. \( \square \)
2. Suppose that each row of an $n \times n$ array $A$ consists of 1's and 0's such that, in any row of $A$, all the 1's come before any 0's in that row. Assuming $A$ is already in memory, describe a method running in $O(n \log n)$ time (not $O(n^2)$ time) for counting the number of 1's in $A$.

Algorithm.

The algorithm is described in Algorithm 1. Essentially, for each row perform a modified binary search to find the index of the first 0. Then, sum up the result of each of the binary search calls to find the total number of 1's.

Algorithm 1 Count 1's

Input: Array $A$ of $n \times n$ 0's and 1's with specified structure

Output: Count of 1's in the matrix

1: $c \leftarrow 0$
2: for $i \leftarrow 1 \ldots n$ do
3: \hspace{1em} $m \leftarrow \text{modifiedBinarySearchForFirst0}()$
4: \hspace{1em} $c \leftarrow c + m$
5: return $c$

Time Complexity.

Theorem 17. Algorithm 1 runs in $O(n \log n)$ time.

Proof. A single binary search runs in $O(\log n)$ time. Thus, $n$ of these searches totals to $O(n \log n)$ time.

Memory Complexity.

Theorem 18. Algorithm 1 uses $O(1)$ additional memory.

Proof. A constant number of temporary variables are used to (1) keep track of binary search indices and (2) the count for our algorithm. Thus, there is $O(1)$ additional memory is required.
3. **Bonus.** The quadratic probing strategy has a clustering problem that relates to the way it looks for open slots when a collision occurs. Namely, when a collision occurs at bucket $h(k)$, we check $A[(h(k) + i^2) \mod N]$, for $i = 1, 2, \ldots, N - 1$.

(a) Show that $i^2 \mod N$ will assume at most $(N+1)/2$ distinct values, for $N$ prime, as $i$ ranges from 1 to $N - 1$. As a part of this justification, note that $i^2 \mod N = (N-i)^2 \mod N$ for all $i$.

(b) A better strategy is to choose a prime $N$, such that $N \mod 4 = 3$ and then to check the buckets $A[(h(k) \pm i^2) \mod N]$ as $i$ ranges from 1 to $(N-1)/2$, alternating between plus and minus. Show that this alternate version is guaranteed to check every bucket in $A$.

**Solution.**

(a) **Theorem 19.** $i^2 \mod N$ will assume at most $(N+1)/2$ distinct values, for $N$ prime, as $i$ ranges from 1 to $N - 1$.

**Proof.** For any given $i$, $i^2 \mod N = (N-i)^2 \mod N$. This implies $1^2 \equiv (N-1)^2 \mod N$, $2^2 \equiv (N-2)^2 \mod N$, \ldots, $(N-1)^2 \equiv 1^2 \mod N$. Thus, the number of distinct values $(N - 1 - 1)/2 \leq (N+1)/2$. \hfill \qed

(b) **Theorem 20.** For a prime $N$, such that $N \mod 4 = 3$, checking buckets $A[(h(k) \pm i^2) \mod N]$ as $i$ ranges from 1 to $(N-1)/2$, alternating between plus and minus is guaranteed to check every bucket in $A$.

**Proof.** Essentially, we need to show that $i^2 \mod N$ will not collide with $-i^2 \mod N$ for all $i$ from 1 to $(N-1)/2$, thus doubling the number of covered buckets in the array. Essentially, we look at the values of $i^2 \mod N \mod 4 = i^2 \mod 3$. In this case, $i^2$ will map to 0 or 1. $-i^2$ however will map to 2 or 3. Essentially, a simple argument of induction shows this. Take the base case as $i = 1$ or $i = 2$. $1^2 \mod N \mod 4 = 1$ and $-1^2 \mod N \mod 4 = N - 1$ mod 4 = 2, and $2^2 \mod N \mod 4 = 0$ and $-2^2 \mod N \mod 4 = N - 4$ mod 4 = 3. Inductive step: Assume this pattern holds for $i$ then we should show it holds for $i + 1$. There are two cases: $i^2 \mod N \mod 4 = 0$ and $i^2 \mod N \mod 4 = 1$. So when it is 0, $i$ is even and $(i+1)^2 \mod N \mod 4 = i^2 + 2i + 1 \mod N \mod 4 = 2i + 1 \mod N \mod 4 = 1$. Similarly when it is 1, $i$ is odd and the modulus comes to 0. Identical patterns prove the negative sides of the pattern as well.