Please note, I made the choice to construct the 3 sets of strings, K and K', every time the main executable is run so that I did not have to store the sets of K and K' in an additional file. However, to ensure these sets were the same for each run of the program, I utilized rand() to choose each letter of the constructed strings and used srand() to set the seed to a specified value before creating each K and corresponding K'. The first set of K and K', which I named K1 and K1', are constructed such that both K1 and K1' each were made up of 10000 strings that were 31 characters in length. For this set, I constructed K1' such that the first 5000 elements were contained within K1, and the last 5000 elements were newly constructed strings that were ensured not to be contained within K1. The second set of K and K', which I named K2 and K2', are constructed such that both K2 and K2' each were made up of 20000 strings that were 31 characters in length. For this set, I constructed K2' such that the last 5000 elements were contained within K2, and the first 15000 elements were newly constructed strings that were ensured not to be contained within K2. The third set of K and K', which I named K3 and K3', are constructed such that both K3 and K3' each were made up of 30000 strings that were 31 characters in length. For this set, I constructed K3' such that the first 12000 elements were contained within K3, and the last 18000 elements were newly constructed strings that were ensured not to be contained within K3. Therefore, the sets have various sizes and various mixtures of "positive" and "negative" keys. These sets of K and K' are used throughout the following tasks. They are initially constructed during task 1 but saved for use in the other tasks.

Task 1

1. For task 1, I implemented bloom filters using the bloom filter library by Arash Patow found at https://github.com/ArashPartow/bloom. I constructed a total of 9 bloom filters, as there were 3 varying sized sets of strings (10000, 20000, and 30000 strings) and 3 different false positive rates to be tested. A bloom filter was created for each combination of these variables where the projected_element_count is set to the corresponding size of the set of strings used and false_positive_probability is set to the corresponding false positive rate. In the executable output, you will see that the bloom filters are numbered using double digits. The first digit indicates the chosen false positive rate (1 = (1/(2^7)), 2 = (1/(2^8)), and 3 = (1/(2^10))) and the second digit indicates the chosen size of the set of strings (1 = 10000, 2 = 20000, and 3 = 30000). After each bloom filter's parameters were set as described above, each element of the corresponding K that matches the size specified to the bloom filter is inserted into the filter. Then, my process of querying involves looping through each element of the corresponding K' that matches the K used and checking if the bloom filter claims the value from K' is contained within it. If the value from K' was from the original K (a "positive key") and is not stated to be contained in the bloom filter, I increment a counter for false negatives as a sanity check. No bloom filter produced false negatives in my tests. If the value from K' was not from the original K (a "negative key") and is stated to be contained in the bloom filter, I increment a counter for false positives. I then calculate the false positive rate by dividing the false positive counter by the number of elements of the corresponding K' that were not in the original K. I then return the false positive rate, time for query, and bloom filter size (calculated using the .size() function in the Arash Patow repo) to the terminal.

2. The programming of this task was fortunately not incredibly difficult due to our ability to use previous implementations of bloom filters. As such, most of the programming was rather repetitive as I ensured that every combination of size of sets and false positive rates was established. One of the main complications I had early on was discovering that the bloom filter was sensitive to the differences between an std::string object and a string literal. I was under the impression that the bloom filter was not correctly inserting my elements from my K sets as I was verifying by using a string literal matching a string from the set K and using the contains() function directly on this literal. This would return false, and I thought there was an issue with the filter itself, but further investigation showed it was an issue matching the types of these strings and the filter worked correctly. Also, during this task, I spent much of my time flushing out the best way to set up the K and K' sets, especially as it centered around ensuring that the randomly generated strings newly made for K' did not accidentally match a string already in K and give us an incorrect false positive as my code would have assumed it should not have been present in the bloom filter.

3. The results of varying the false positive rates and sets of strings are represented in the table below, where the columns represent the various target false positive rates, and the rows represent the various K and K' sets described at the top of the document.

	target false positive rate (1/(2^7))	**target false positive rate (1/(2^8))**	**target false positive rate (1/(2^10))**
K1 and K1' (10000 strings, 50% positive 50% negative)	**Observed false positive rate:** 0.0086 **Time to Query K':** 2014684 nanoseconds **Size of Bloom Filter:** 100992 bits	**Observed false positive rate:** 0.0052 **Time to Query K':** 2248399 nanoseconds **Size of Bloom Filter:** 115416 bits	**Observed false positive rate:** 0.0006 **Time to Query K':** 9607370 nanoseconds **Size of Bloom Filter:** 144272 bits
K2 and K2' (20000 strings, 25% positive 75% negative)	**Observed false positive rate:** 0.01 **Time to Query K':** 2111629 nanoseconds **Size of Bloom Filter:** 201984 bits	**Observed false positive rate:** 0.00413333 **Time to Query K':** 3438705 nanoseconds **Size of Bloom Filter:** 230832 bits	**Observed false positive rate:** 0.000933333 **Time to Query K':** 4670956 nanoseconds **Size of Bloom Filter:** 288544 bits
K3 and K3' (30000 strings, 40% positive 60% negative)	**Observed false positive rate:** 0.00827778 **Time to Query K':**	**Observed false positive rate:** 0.00388889 **Time to Query K':**	**Observed false positive rate:** 0.00144444 **Time to Query K':**

	5891537 nanoseconds **Size of Bloom Filter:** 302968 bits	8715620 nanoseconds **Size of Bloom Filter:** 346248 bits	6993642 nanoseconds **Size of Bloom Filter:** 432808 bits

As we expected, we see that for each of the sets, as the target false positive rates decreased, the size of the bloom filter increases. Also, as the size of the sets of K and K' increased, we saw the size of the bloom filter increase as well. Additionally, note that for these sets and bloom filters, the observed false positive rates did not match the target false positive rate perfectly. However, for most cases, the observed false positive rates were similar to the target or on the same order of the target rate. Also, note that with a smaller target false positive rate (and subsequently a smaller observed false positive rate as the observed rate always decreased when the target rate decreased), the lookup time for the query increased. Therefore, there is a tradeoff between look up time and false positive rate.

Task 2

1. For task 2, I implemented the minimal perfect hash functions (MPHFs) using BBHash for C++ found here https://github.com/rizkg/BBHash/tree/alltypes. Note, I specifically use the alltypes branch as it was stated that it was best suited for use with strings. My implementation heavily followed the example provided by the BBHash repo for strings found here https://github.com/rizkg/BBHash/blob/alltypes/example_custom_hash_strings.cpp. I created the Custom_string_Hasher class and told BBhash to use this custom hash as instructed in the example. I then made 3 minimal perfect hash functions, one for each set of K and K' specified above. Each of these 3 MPHFs has a "number of elements" parameter set to the size of its corresponding K set, the "data input" set to its corresponding K set, 1 thread, and a gamma factor of 2 which is the default suggested by the BBHash repository. These are all supplied in a constructor which sets up the minimal perfect hash function using the elements in the corresponding K set. Then, my process of querying involves looping through each element of the corresponding K' that matches the K used and checking if the MPHF produces an index for the value from K'. If the value from K' was from the original K (a "positive key") and the MPHF returns ULLONG_MAX (the value stated by BBHash to signify the MPHF does not contain the value in the set) rather than an index, I increment a counter for false negatives as a sanity check. No MPHF produced false negatives in my tests. If the value from K' was not from the original K (a "negative key") and the MPHF returns an index rather than ULLONG_MAX, I increment a counter for false positives. I then calculate the false positive rate by dividing the false positive counter by the number of elements of the corresponding K' that were not in the original K. I then return the false positive rate, time for query, and MPHF size (which is calculated and printed to the terminal during the construction of the MPHF due to the nature of BBHash) to the terminal.

2. Again, as this task allowed us to use outside implementations of minimal perfect hash, the programming of this task implementation was not incredibly difficult. BBHash was rather intuitive after reading the example documents. So, the most difficult part to me was just understanding the effect that gamma had on the MPHFs. I determined that increasing the gamma factor significantly decreased the number of false positives produced, but also caused a significant increase in the size of the MPHF. As such, I decided it would be best to leave the gamma factor at 2 as I agreed with the sample code that it created a fair tradeoff.

3. The results of created a MPHF for varying sized sets of strings are represented in the table below.

K1 and K1' **(10000 strings, 50% positive 50% negative)**	**K2 and K2'** **(20000 strings, 25% positive 75% negative)**	**K3 and K3'** **(30000 strings, 40% positive 60% negative)**
Observed false positive rate: 0.907 **Time to Query K':** 3715544 nanoseconds **Size of MPHF:** 43136 bits	**Observed false positive rate:** 0.9092 **Time to Query K':** 8092845 nanoseconds **Size of MPHF:** 79936 bits	**Observed false positive rate:** 0.969 **Time to Query K':** 12020059 nanoseconds **Size of MPHF:** 117248 bits

The size of the MPHFs are roughly 40% of the size of the smallest bloom filters constructed above for each set of K and K'. However, the lookup times and false positive rates appear to be worse than those of the smallest bloom filters. Of particular concern is the fact that there appears to be at least a 90% false positive rate for each MPHF, whereas the worst observed false positive rate for the bloom filters was 1%. However, I had no expectations that the MPHFs would provide low false positive rates as they are not AMQ's and are not designed for that purpose.

Task 3:

1. For task 3, I relied on the work done in task 2 for the MPHF portion as they were in the same ./src/main.cpp file and I would not need to create these structures again. For the fingerprint array portion, I made use of the std::hash<std::string>{}() function and int_vector objects from the sdsl-lite library. I constructed a total of 9 fingerprint arrays, as there were 3 varying sized sets of strings (10000, 20000, and 30000 strings) and 3 different fingerprint sizes to be tested. A fingerprint array (in the form of a sdsl int_vector) was created for each combination of these variables where the size of the int_vector is set to the corresponding size of the set of strings used and width of the int_vector is set to the corresponding fingerprint size. In the executable output, you will see that the combinations of MPHF and fingerprint arrays are numbered using double digits. The first digit indicates the chosen fingerprint size (1 = 7, 2 = 8, and 3 = 10) and the second digit indicates the chosen size of the set of strings (1 = 10000, 2 = 20000, and 3 = 30000). For each individual combination of MPHF and fingerprint array, the

int_vector for the fingerprint array is initialized with the parameters above, then a for loop is run to traverse over the set K that corresponds to the size set selected for the given combination. In this for loop, the index is retrieved from the MPHF that corresponds to the K for each element in K. Then the hash function is called on the element, the specified number of bits are taken from the end of resulting value and stored in the int_vector at the previously determined index for that element. Then, my process of querying involves looping through each element of the corresponding K' that matches the K used and checking if the MPHF and fingerprint array claims the value from K' is contained within it. If the value from K' was from the original K (a "positive key") and the MPHF returns ULLONG_MAX rather than an index or the last bits of the hashed value of this key does not match the value stored at the specified index in the fingerprint array, I increment a counter for false negatives as a sanity check. No combination produced false negatives in my tests. If the value from K' was not from the original K (a "negative key") and the MPHF returns an index rather than ULLONG_MAX and the last bits of the hashed value of this key match the value stored at the specified index, I increment a counter for false positives. I then calculate the false positive rate by dividing the false positive counter by the number of elements of the corresponding K' that were not in the original K. I then return the false positive rate, time for query, and the size of the int_vector (calculated using size_in_bytes() from sdsl).

2. It was a bit difficult to select a "good" hash function, but I decided the built in std::hash function would serve my purposes well enough. The most difficult part of this task was keeping some of the elements of the combination straight in order to ensure the correct values were used. This included using the correct MPHF for the combination, remembering each condition necessary for determining false negatives or positives, and determining the correct value to perform a "bitwise and" with so that only the proper number of bits were taken from the end of the hashed value.

3. The results of varying the fingerprint sizes and sets of strings are represented in the table below, where the columns represent the various fingerprint sizes, and the rows represent the various K and K' sets described at the top of the document.

	fingerprints consisting of 7 bits	**fingerprints consisting of 8 bits**	**fingerprints consisting of 10 bits**
K1 and K1' (10000 strings, 50% positive 50% negative)	**Observed false positive rate:** 0.2574 **Time to Query K':** 4803750 nanoseconds **Size of MPHF + Fingerprint Arr:** 113224 bits	**Observed false positive rate:** 0.123 **Time to Query K':** 4949584 nanoseconds **Size of MPHF + Fingerprint Arr:** 123208 bits	**Observed false positive rate:** 0.031 **Time to Query K':** 4607923 nanoseconds **Size of MPHF + Fingerprint Arr:** 143240 bits

K2 and K2' (20000 strings, 25% positive 75% negative)	**Observed false positive rate:** 0.356733 **Time to Query K':** 10561610 nanoseconds **Size of MPHF + Fingerprint Arr:** 220040 bits	**Observed false positive rate:** 0.1758 **Time to Query K':** 10201768 nanoseconds **Size of MPHF + Fingerprint Arr:** 240008 bits	**Observed false positive rate:** 0.0445333 **Time to Query K':** 10349906 nanoseconds **Size of MPHF + Fingerprint Arr:** 280008 bits
K3 and K3' (30000 strings, 40% positive 60% negative)	**Observed false positive rate:** 0.413444 **Time to Query K':** 14848462 nanoseconds **Size of MPHF + Fingerprint Arr:** 327368 bits	**Observed false positive rate:** 0.206778 **Time to Query K':** 15184371 nanoseconds **Size of MPHF + Fingerprint Arr:** 357320 bits	**Observed false positive rate:** 0.0507778 **Time to Query K':** 33887502 nanoseconds **Size of MPHF + Fingerprint Arr:** 417352 bits

As theorized, we see a decrease in the observed false positive rate when using a combination of the MPHFs and fingerprint arrays. For each set K and K', we can see that using MPHF and fingerprint arrays leads to at least a 50% decrease in the false positive rate of using MPHF alone. Also, as we would expect, the combined size of the MPHF and int_vectors used were larger than the MPHF alone. We see that the increase is approximately equal to (the size of the corresponding K)*(fingerprint size) plus some bits for overhead. We can see that the query time is greater than that of MPHF alone as well, which makes sense as we are doing extra lookups for each element in the query. Note, that though the false positive rate did improve, it did not perfectly align with the target rates of (1/(2^7)), (1/(2^8)), and (1/(2^10)) for the fingerprint sizes 7, 8, and 10 respectively. Maybe if a better hash function was used, this could have been better realized. I also know that increasing the gamma factor of the MPHF could have brought that rate down as well. However, I will point out that the false positive rates were reduced from one fingerprint size to the next as we expected. The false positive rates of each set K and K' reduced by approximately half from fingerprint size 7 to 8, and the false positive rates of each set K and K' reduced by approximately 3 quarters from fingerprint size 8 to 10. Finally, except in one outlier case, it appears that lookup time was not as heavily impacted by the false positive rates as we saw with bloom filters. For each set K and K', the lookup time across each fingerprint size was approximately the same.

SAMPLE OUTPUT FROM EXECUTABLE

----------Task1: Bloom Filters----------
Bloom filter 11. FP Rate entered (1/(2^7)). Sets size 10000. String size 31.
K1' is 50% keys present in K1 and 50% keys not in K1.
BF11 contains 0 false negatives
BF11 contains 43 false positives with rate 0.0086
Time taken for querying K1': 2014684 nanoseconds
Size of bloom filter: 100992 bits

Bloom filter 12. FP Rate entered (1/(2^7)). Sets size 20000. String size 31.
K2' is 25% keys present in K2 and 75% keys not in K2.
BF12 contains 0 false negatives
BF12 contains 150 false positives with rate 0.01
Time taken for querying K2': 2111629 nanoseconds
Size of bloom filter: 201984 bits

Bloom filter 13. FP Rate entered (1/(2^7)). Sets size 30000. String size 31.
K3' is 40% keys present in K3 and 60% keys not in K3.
BF13 contains 0 false negatives
BF13 contains 149 false positives with rate 0.00827778
Time taken for querying K3': 5891537 nanoseconds
Size of bloom filter: 302968 bits

Bloom filter 21. FP Rate entered (1/(2^8)). Sets size 10000. String size 31.
K1' is 50% keys present in K1 and 50% keys not in K1.
BF21 contains 0 false negatives
BF21 contains 26 false positives with rate 0.0052
Time taken for querying K1': 2248399 nanoseconds
Size of bloom filter: 115416 bits

Bloom filter 22. FP Rate entered (1/(2^8)). Sets size 20000. String size 31.
K2' is 25% keys present in K2 and 75% keys not in K2.
BF22 contains 0 false negatives
BF22 contains 62 false positives with rate 0.00413333
Time taken for querying K2': 3438705 nanoseconds
Size of bloom filter: 230832 bits

Bloom filter 23. FP Rate entered (1/(2^8)). Sets size 30000. String size 31.
K3' is 40% keys present in K3 and 60% keys not in K3.
BF23 contains 0 false negatives
BF23 contains 70 false positives with rate 0.00388889
Time taken for querying K3': 8715620 nanoseconds
Size of bloom filter: 346248 bits

Bloom filter 31. FP Rate entered (1/(2^10)). Sets size 10000. String size 31.
K1' is 50% keys present in K1 and 50% keys not in K1.
BF31 contains 0 false negatives
BF31 contains 3 false positives with rate 0.0006
Time taken for querying K1': 9607370 nanoseconds
Size of bloom filter: 144272 bits

Bloom filter 32. FP Rate entered (1/(2^10)). Sets size 20000. String size 31.
K2' is 25% keys present in K2 and 75% keys not in K2.
BF32 contains 0 false negatives
BF32 contains 14 false positives with rate 0.000933333
Time taken for querying K2': 4670956 nanoseconds
Size of bloom filter: 288544 bits

Bloom filter 33. FP Rate entered (1/(2^10)). Sets size 30000. String size 31.
K3' is 40% keys present in K3 and 60% keys not in K3.
BF33 contains 0 false negatives
BF33 contains 26 false positives with rate 0.00144444
Time taken for querying K3': 6993642 nanoseconds
Size of bloom filter: 432808 bits

----------Task2: MPHF----------
MPHF 1. Sets size 10000. String size 31. Gamma 2.
K1' is the same K1' as above, it is 50% keys present in K1 and 50% keys not in K1.
[Building BooPHF] 100 % elapsed: 0 min 0 sec remaining: 0 min 0 sec
Bitarray 43136 bits (100.00 %) (array + ranks)
final hash 0 bits (0.00 %) (nb in final hash 0)
boophf bits/elem : 4.313600
MPHF1 contains 0 false negatives
MPHF1 contains 4535 false positives with rate 0.907
Time taken for querying K1': 3715544 nanoseconds

MPHF 2. Sets size 20000. String size 31. Gamma 2.
K2' is the same K2' as above, it is 25% keys present in K2 and 75% keys not in K2.
[Building BooPHF] 100 % elapsed: 0 min 0 sec remaining: 0 min 0 sec
Bitarray 79936 bits (100.00 %) (array + ranks)
final hash 0 bits (0.00 %) (nb in final hash 0)
boophf bits/elem : 3.996800
MPHF2 contains 0 false negatives
MPHF2 contains 13638 false positives with rate 0.9092
Time taken for querying K2': 8092845 nanoseconds

MPHF 3. Sets size 30000. String size 31. Gamma 2.

K3' is the same K3' as above, it is 40% keys present in K3 and 60% keys not in K3.
[Building BooPHF] 100 % elapsed: 0 min 0 sec remaining: 0 min 0 sec
Bitarray 117248 bits (100.00 %) (array + ranks)
final hash 0 bits (0.00 %) (nb in final hash 0)
boophf bits/elem : 3.908267
MPHF3 contains 0 false negatives
MPHF3 contains 17442 false positives with rate 0.969
Time taken for querying K3': 12020059 nanoseconds

----------Task3: Fingerprint Array----------
MPHF + FP 11. Fingerprint size 7. Sets size 10000. String size 31. Gamma 2.
K1' is the same K1' as above, it is 50% keys present in K1 and 50% keys not in K1.
FP11 contains 0 false negatives
FP11 contains 1287 false positives with rate 0.2574
Time taken for querying K1': 4803750 nanoseconds
Size of fingerprint array alone: 70088 bits. Look at MPHF section for its size.

MPHF + FP 12. Fingerprint size 7. Sets size 20000. String size 31. Gamma 2.
K2' is the same K2' as above, it is 25% keys present in K2 and 75% keys not in K2.
FP12 contains 0 false negatives
FP12 contains 5351 false positives with rate 0.356733
Time taken for querying K2': 10561610 nanoseconds
Size of fingerprint array alone: 140104 bits. Look at MPHF section for its size.

MPHF + FP 13. Fingerprint size 7. Sets size 30000. String size 31. Gamma 2.
K3' is the same K3' as above, it is 40% keys present in K3 and 60% keys not in K3.
FP13 contains 0 false negatives
FP13 contains 7442 false positives with rate 0.413444
Time taken for querying K3': 14848462 nanoseconds
Size of fingerprint array alone: 210120 bits. Look at MPHF section for its size.

MPHF + FP 21. Fingerprint size 8. Sets size 10000. String size 31. Gamma 2.
K1' is the same K1' as above, it is 50% keys present in K1 and 50% keys not in K1.
FP21 contains 0 false negatives
FP21 contains 615 false positives with rate 0.123
Time taken for querying K1': 4949584 nanoseconds
Size of fingerprint array alone: 80072 bits. Look at MPHF section for its size.

MPHF + FP 22. Fingerprint size 8. Sets size 20000. String size 31. Gamma 2.
K2' is the same K2' as above, it is 25% keys present in K2 and 75% keys not in K2.
FP22 contains 0 false negatives
FP22 contains 2637 false positives with rate 0.1758
Time taken for querying K2': 10201768 nanoseconds

Size of fingerprint array alone: 160072 bits. Look at MPHF section for its size.

MPHF + FP 23. Fingerprint size 8. Sets size 30000. String size 31. Gamma 2.
K3' is the same K3' as above, it is 40% keys present in K3 and 60% keys not in K3.
FP23 contains 0 false negatives
FP23 contains 3722 false positives with rate 0.206778
Time taken for querying K3': 15184371 nanoseconds
Size of fingerprint array alone: 240072 bits. Look at MPHF section for its size.

MPHF + FP 31. Fingerprint size 10. Sets size 10000. String size 31. Gamma 2.
K1' is the same K1' as above, it is 50% keys present in K1 and 50% keys not in K1.
FP31 contains 0 false negatives
FP31 contains 155 false positives with rate 0.031
Time taken for querying K1': 4607923 nanoseconds
Size of fingerprint array alone: 100104 bits. Look at MPHF section for its size.

MPHF + FP 32. Fingerprint size 10. Sets size 20000. String size 31. Gamma 2.
K2' is the same K2' as above, it is 25% keys present in K2 and 75% keys not in K2.
FP32 contains 0 false negatives
FP32 contains 668 false positives with rate 0.0445333
Time taken for querying K2': 10349906 nanoseconds
Size of fingerprint array alone: 200072 bits. Look at MPHF section for its size.

MPHF + FP 33. Fingerprint size 10. Sets size 30000. String size 31. Gamma 2.
K3' is the same K3' as above, it is 40% keys present in K3 and 60% keys not in K3.
FP33 contains 0 false negatives
FP33 contains 914 false positives with rate 0.0507778
Time taken for querying K3': 33887502 nanoseconds
Size of fingerprint array alone: 300104 bits. Look at MPHF section for its size.