# Problem 1:

**Is It A Rotation Of A Palindrome?**

Given a string, check if it is a rotation of some palindromic string or not.

**Example**

Input: “aab”

Output: 1

“aab” is a rotation of the palindromic string “aba”. Rotating “aba” to the right gives “aab”.

Output: Return 1 if the input string is a rotation of some palindromic string else return 0.

Constraints:

* 1 <= length of the input string <= 4000
* ‘a’ <= any character in the string <= ‘z’

# Solution

We provided one solution that takes O(N^2) time. It is something one could realistically come up with and code on an interview.

There exist more efficient solutions. Using [suffix array](http://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/) and [LCP array](http://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/) makes O(N\*log(N)) possible. Using the so-called “[longest palindromic substring](https://en.wikipedia.org/wiki/Longest_palindromic_substring)” makes O(N) possible.

For a string of length N, the number of distinct strings obtained by rotating it is <= N (some rotations might duplicate, that’s why the total number can be less than N). Initially one might think there are many ways to apply rotations, such as left rotation -> left rotation -> left rotation -> right rotation and so on. But a left rotation followed by a right rotation would cancel out each other. Also applying left shift x times is the same as applying left shift by x % N times and applying right shift x times is the same as applying right shift by x % N times. For better understanding try some examples.

If a string x is a rotation of a palindromic string p, then at least one rotation of x equals to p. So we can try all N rotations of the given string and check if any rotation is a palindrome. Finding if a given string is a palindrome takes O(N) time.

**Time Complexity:**

O(N) \* O(N) = O(N ^ 2).

**Auxiliary Space Used:**

O(1).

**Space Complexity:**

O(N).

# Problem 2:

**Detect Prime Numbers**

Given an array of integers, check each number if it’s prime.

**Example**

Input: [6, 2, 5, 8]

Output: “0110”

6 is not a prime number. (6 = 2 \* 3)

2 is a prime number.

5 is a prime number.

8 is not a prime number. (8 = 2 \* 4)

**Notes**

Input Parameters: Function has one argument, an integer array.

Output: Return a string of the length equal to the size of the input array. For each number in the array the string has to have either “0” (not prime) or “1” (prime) character.

Constraints:

* 1 <= any input array element <= 4 \* 10^6
* 1 <= number of array elements <= 3 \* 10^5

Output Format: There will be one line, containing a resultant string *res*. For input *N* = 4 and *a* = [6, 2, 5, 8], output will be:

0110

# Solution

We provided three sample solutions plus some notes in the very end.

**1. brute\_force\_solution.cpp**

For any number a[i], we iterate over 2 to a[i] - 1 and check if any of them divides a[i] or not. If we find at least one number that divides a[i] then a[i] is not a prime number.

**Time Complexity:**

O(N \* MAX\_A).

**Auxiliary Space Used:**

O(1).

**Space Complexity:**

O(N).

**2. sub\_optimal\_solution.cpp**

Any positive number x, which is non-prime (and not 1), can be written as x = a \* b where a > 1 and b > 1. Here both a and b can not be > root(x), because if a > root(x) and b > root(x) then, a \* b > root(x) \* root(x), hence a \* b > x, which contradicts a \* b = x. When number is a square like 16 then it can be written as root(16) \* root(16). So, non-prime number x can be written as a \* b having at least one of them <= root(x). Now in the previous solution for each a[i], loop from root(a[i]) + 1 to a[i] - 1 is not necessary. Time complexity improves.

**Time Complexity:**

O(N \* root(MAX\_A)).

**Auxiliary Space Used:**

O(1).

**Space Complexity:**

O(N).

**3. optimal\_solution.cpp**

Still we can improve. We can write any composite number as multiplication of prime numbers. Like 60 = 2 \* 2 \* 3 \* 5. So, when we find any prime number, we can visit all the multiple of it and mark them as visited (non-prime). We start from 2 as base case, consider 2 as prime number and mark all its multiples 4, 6, 8, 10, ... as visited (non-prime) because they are multiples of 2.

After considering 2, we will consider 3. Now, 3 is not marked as visited, which means, there are no factors of 3 and it is a prime number. Hence for 3 also, we do the same thing, mark all its multiples 6, 9, 12, 15 ... as visited (non-prime), because they are multiple of 3.

Now 4 comes, but 4 is marked as visited by 2, so we know that 4 is not a prime number. Here also we can do the same thing: mark all its multiples 8, 12, 16 ... as visited, but all those positions are already marked as visited by 2, because all multiples of 4 are also multiples of 2. So, no need to do rework! Now, we keep on following the same steps for next numbers. Try to find prime numbers till 30 and you will get a more clear idea.

Still there can be some optimizations. When we find any prime number like 11, we mark 22, 33. 44, 55, 66 ... them as non-prime, but 22, 33, 44, 55 are already visited by smaller prime numbers. Like 22 is visited by 2, 33 is visited by 3, 44 is visited by 4 ... 110 is visited by 10. So, instead of starting marking multiples as non-prime from x+x, we can start from x\*x.

This algorithm is called "Sieve of Eratosthenes".

**Time Complexity:**

O(N \* log(log MAX\_A)).

**Auxiliary Space Used:**

O(MAX\_A).

**Space Complexity:**

O(N).

**General notes and comparison of algorithms**

Can we say that solution with time complexity O(N \* log(log MAX\_A)) is always better than solution with time complexity O(N \* root(MAX\_A))? No. It depends on the situation. In terms of time complexity of course it is better, but we should also consider space! Solution with time complexity O(N \* root(A)) requires O(1) extra space, but other needs O(MAX\_A) extra space. So, when space is more important than time, we should opt for the slower one.

If given a single integer and need to find if it is a prime or not, should we use Sieve of Eratosthenes? Probably not. We can check one number in O(root(A)) by iterating over 2 to root(A).

Sieve of Eratosthenes algorithm is helpful when

a) many numbers need to be checked and so the pre-processing can help and

b) if those numbers are limited to a given range - so that we can estimate how much space the algorithm would use.

In this problem we are given the range for the numbers. But when given a stream of random numbers and nothing is specified about range of a[i] then we can use caching. Caching will be useful only when some numbers are going to repeat, though in real life that is likely to happen. We maintain two hash-tables. One containing prime numbers encountered until now and other containing non-prime numbers encountered. For each number, we check the presence in our hash tables and if it is present in one, we are done. If it is not present in either however, we use the O(root(number)) method to check if it is prime and add it into the appropriate hash table for future reference.

To further improve auxiliary space, we can check even numbers without adding them to the hashmap. Other optimizations can be considered in a real life implementation.

# Problem 3

**Divide An Integer By Another Integer**

Given two integers, find their quotient, i.e. the integer result of dividing the first by the second.

**Example**

Input: 5, 2

Output: 2

**Notes**

Input Parameters: Two integers *a* and *b*.

Output: An integer quotient of *a* and *b*.

Constraints:

* -10^18 <= *a*, *b* <= 10^18
* *b* != 0
* You are not allowed to use division (/) operator.
* You are not allowed to use multiplication (\*) operator.
* You are not allowed to use mod (%) operator.

# Solution

Solution with time complexity O(a / b) and space complexity O(1) can be achieved using addition or subtraction.

We can divide our problem in four parts:

1) a >= 0 and b > 0

2) a < 0 and b > 0

3) a >= 0 and b < 0

4) a < 0 and b < 0

When a >= 0 and b > 0 then we can do something like this:

sum = 0

q = 0

while (sum + b <= a)

sum += b

q++

return q

Some modification in the above code will also work with other combinations. But still we can improve time complexity.

Let's take a and b such that a % b = 0 so we can write q = a / b. q \* b = a. Let’s think about the binary representation of the numbers.

q \* b = q

(q31q30...q0) \* b = a (in the binary representation)

(2^31 \* q31 + 2^30 \* q30 + ... + 2^0 \* q0) \* b = a

To find the value for each bit we can start from the left side.

First we try to set q31 = 1, if 2^31 \* b <= a then we set q31 = 1, but if 2^31 \* b > a then we set q31 = 0.

When we set q31 = 0 then we have to solve

(2^30 \* q30 + ... + 2^0 \* q0) \* b = a

and when we set q31 = 1 then we have to solve

(2^30 \* q30 + ... + 2^0 \* q0) \* b = a - 2^31 \* b.

Consider 37 / 3. We keep on shifting the divisor by 1 binary position (that multiplies it by 2) until it exceeds 37. Here it will be 3 -> 6 -> 12 -> 24. Now we can write our division 37 / 3 = (37 / (3 \* 8)) \* 8 + (37 - (3 \* 8)) / 3. Now first part is (37 / 24) \* 8 = 1 \* 8 = 1 \* 2^(number of shifts). Second part is 13 / 3 and it is a smaller version of the original problem.

All three solutions we provided are different implementations of the idea we discussed here.

**1) other\_solution1.cpp: Recursive solution.**

**Time Complexity:**

O(log(a) ^ 2). Recursive function can be called O(log(a)) times, and in each function call we are shifting no\_of\_shifts times that is O(log(a)). Shift operation takes O(1) time.

**Auxiliary Space Used:**

O(log(a)) due to the recursive function call.

**Space Complexity:**

O(log(a))

**2) other\_solution2.cpp: Iterative solution.**

**Time Complexity:**

O(log(a)^2).

**Auxiliary Space Used:**

O(1).

**Space Complexity:**

O(1).

**3) optimal\_solution.cpp: Iterative solution.**

After some observations in other\_solution2.cpp we can notice that no\_of\_shifts always decreases. Suppose it decreases 60 -> 55 -> 50 -> ... -> 0. Then in other\_solution2.cpp we start no\_of\_shifts from 0 and increment it to 60, again for 55 we will increment no\_of\_shifts from 0 to 55. But as we know it will decrease from 60 to 55, we can directly start from 60 and quickly reach 55. Then from 55 to 50 (instead of 0 to 55)... After this optimization, O(log(a)) shifts remain. Shift operation takes O(1) time.

If you use C, replace abs() calls by fabs() after copying our solution in C++.

**Time Complexity:**

O(log(a) + log(a)) = O(log(a)).

**Auxiliary Space Used:**

O(1).

**Space Complexity:**

O(1).