a. The following function has a foreach loop  
int ArraySum(const vector<int>& arr) {

O(1)

O(1)

int sum = 0;

for (int num : arr) {  
sum += num;  
}  
return sum;  
}

Space complexity = O(1) + O(1) = O(1)

b. Assume str1 has m characters and str2 has n characters  
string StringConcatenation(const string& str1,  
const string& str2) {  
return str1 + str2;

O(m + n)

}

Space complexity = O(m + n)

c. Assume matrix is a 2D matrix with dimensions m x n  
vector<vector<int>> TransposeMatrix(const  
vector<vector<int>>& matrix) {  
int m = matrix.size();  
int n = matrix[0].size();  
vector<vector<int>> transpose(n,  
vector<int>(m));  
for (int i = 0; i < m; i++) {  
for (int j = 0; j < n; j++) {  
transpose[j][i] = matrix[i][j];  
}  
}  
return transpose;  
}

O(m \* n)

O(1)

O(1)

Space complexity = O(1) + O(1) + O(m \* n) = O(mn)

vector<int> SieveOfEratosthenes(int n) { //Find n Prime numbers

O(n + 1)

vector<bool> primesCheck(n + 1, true);

vector<int> primesList(n);

O(n)

primesCheck[0] = false;

primesCheck[1] = false;

if (n <= 0)

return primesList;

O(1)

int curNum = 2;

int listIndex = 0;

O(1)

while (listIndex != n) {

primesList[listIndex++] = curNum;

for (int i = curNum \* curNum; i < primesCheck.size(); i += curNum){ //start from curNum \* curNum as multiples of num that less than curNum are tested

O(1) for i

if(primesCheck[i])

primesCheck[i] = false;

}

while (++curNum < primesCheck.size() && !primesCheck[curNum]) {} //increment curNum to next Primes

//same as before, multiples of num that less than curNum are all tested

//e.g. if curNum is 11, 2 \* 11, 3 \* 11 .... are tested, I just need to start as 11 \* 11

// if the test list is 2 to 100, as 11 \* 11 > 100, it means 100 = a \* b, a > 11 and b < 11, and b are tested

// this rule also applicable to numbers that less than 100, so the remaining number in the checklist are all Primes

if (curNum \* curNum >= primesCheck.size()) {

while (curNum < primesCheck.size() && listIndex < n) {

if (primesCheck[curNum])

primesList[listIndex++] = curNum;

curNum++;

}

if (listIndex == n)

return primesList;

else {

O(1)

int oldSize = primesCheck.size();

O(n) for first time, O(2n) for second time, O(4n) …

primesCheck.resize(oldSize \* 2, true);

for (int j = 0; j < listIndex; j++) {

O(1) for j

O(1)

int num = oldSize;

while (num % primesList[j] != 0) //Use ++ to find divisable then reassign i to that

num++;

for (int k = num; k < primesCheck.size(); k += primesList[j]) {

if(primesCheck[k])

O(1) for k

primesCheck[k] = false;

}

}

while (++curNum < primesCheck.size() && !primesCheck[curNum]) {} //increment curNum to next Primes

}

}

}

primesList[listIndex] = curNum;

return primesList;

}

Space complexity = O(n + 1) + O(n) + O(1) + O(1) + O(1) + O(1) + O(n) [may become O(n^2) or even more, depending on Prime Number distribution] + O(1) + O(1) + O(1) = O(n) [may become O(n^2) or even more, depending on Prime Number distribution]