Compute the time complexity of the given numbers, identify the correct complexity and show your solutions and answer. Failure to will not gain any mark on this activity. Screen shot your solution and answer, box your answer.

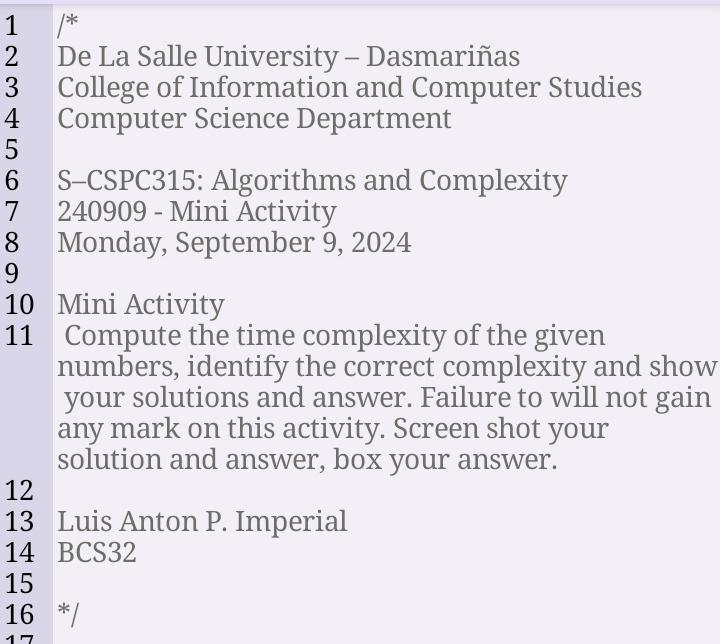
**View raw file of answers at:** https://dlsudphl-my.sharepoint.com/:u:/g/personal/ilp0824\_dlsud\_edu\_ph/ERJ28WdOGHxFoi1B86dYmHEBNpInG7DB6oyeqj-HqCOrEQ?e=rfCht7

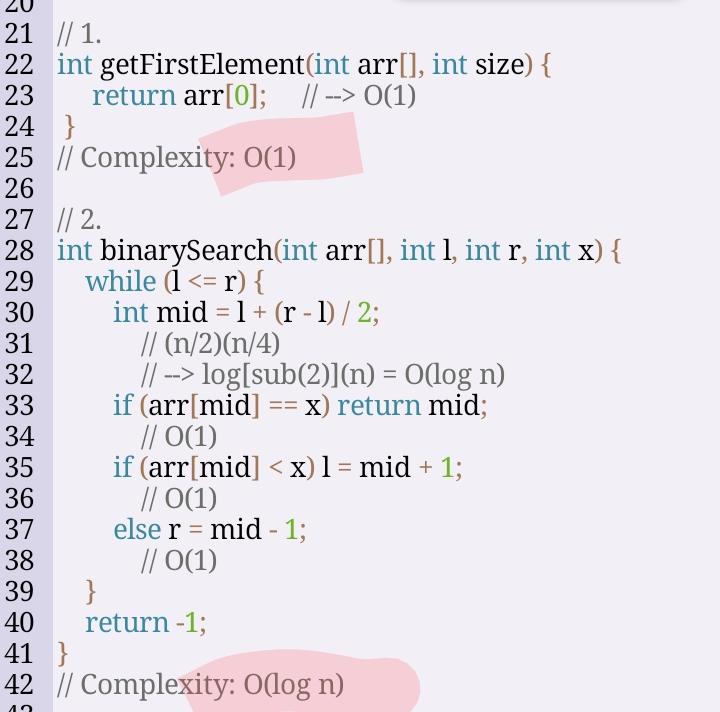
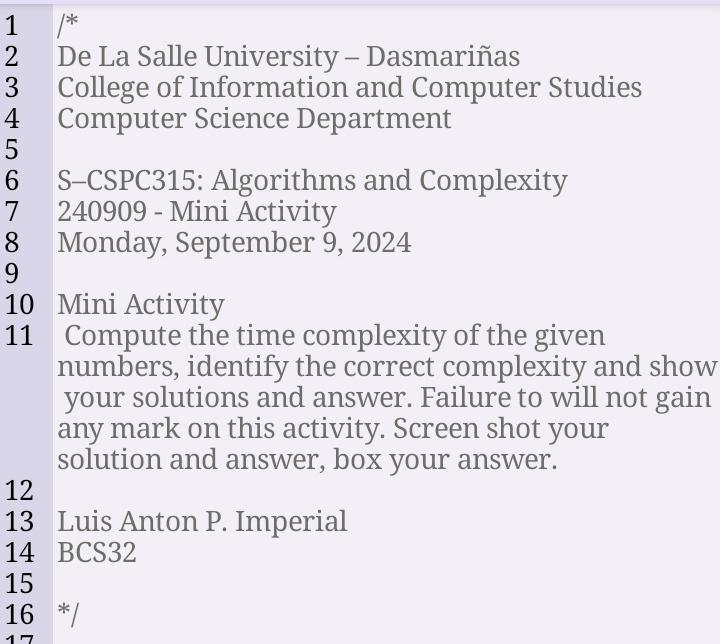
# Table of Contents

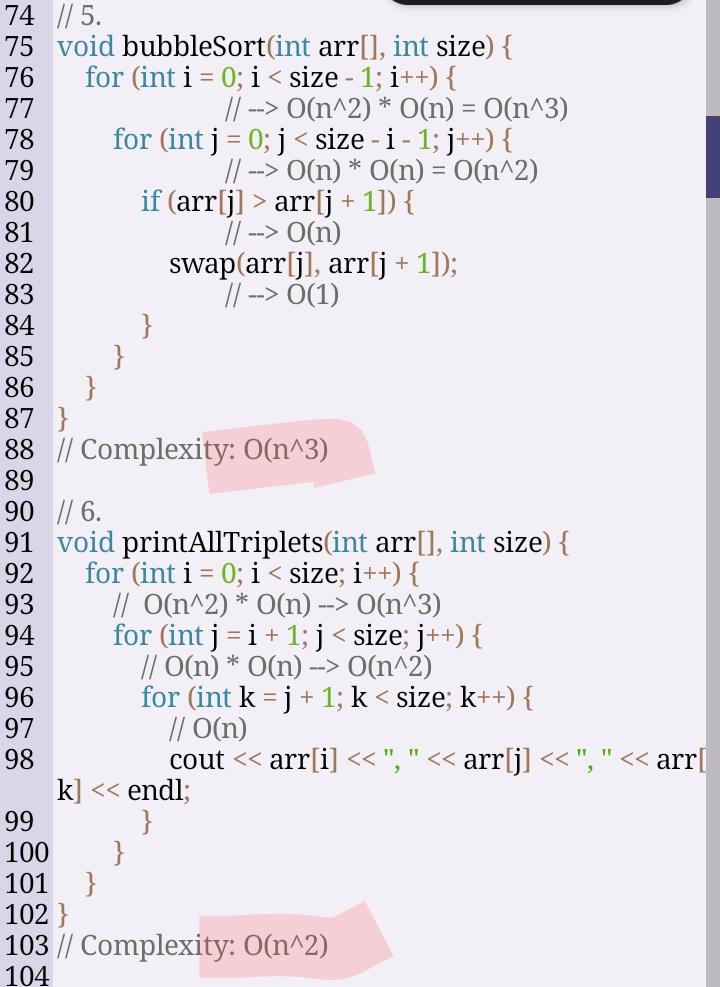
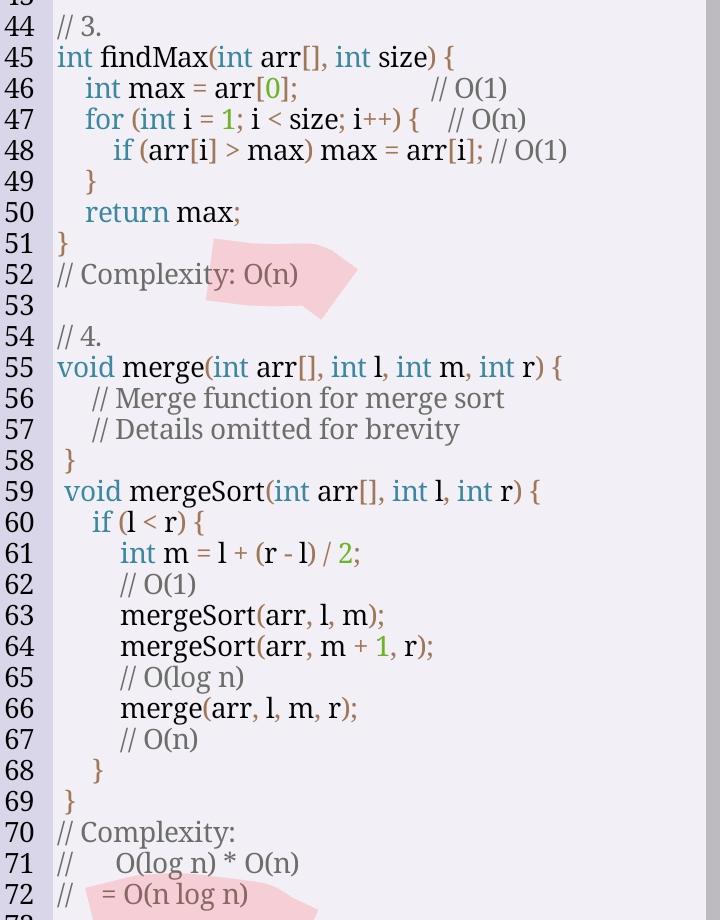
1. Answers
   1. In screenshot form, with final answer boxed in red
   2. In raw form

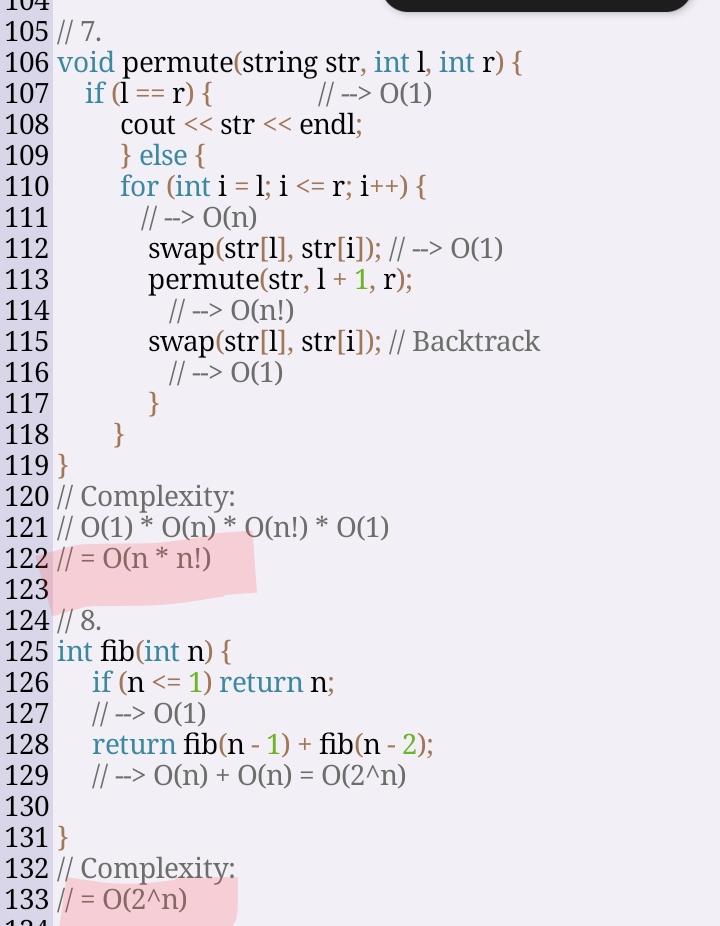
# Answers

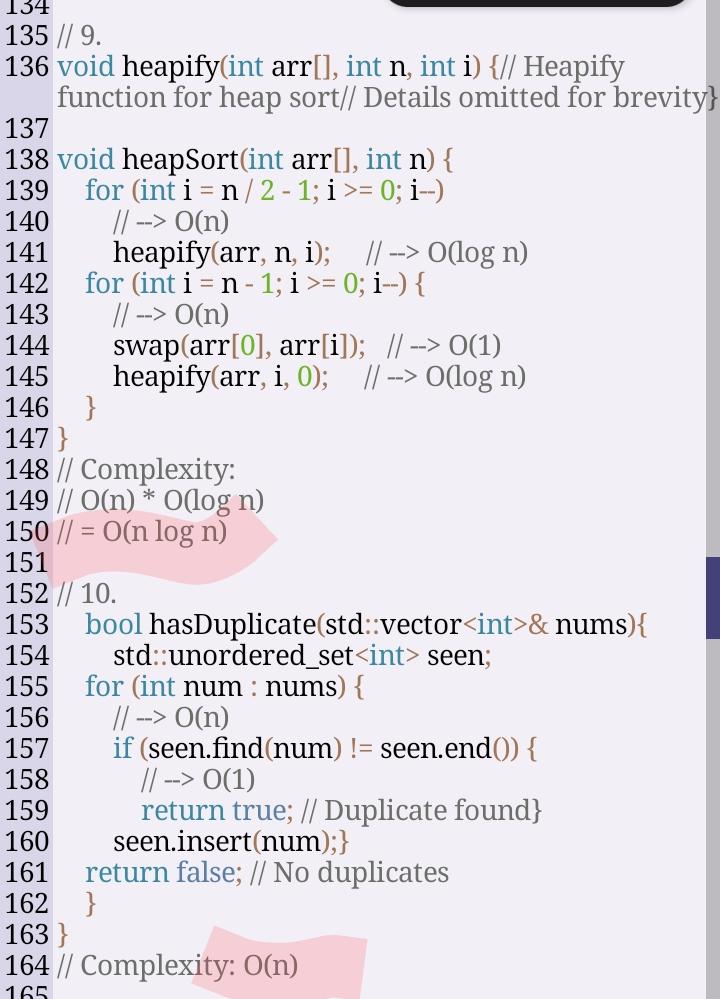
## In screenshot form, with final answers boxed in red

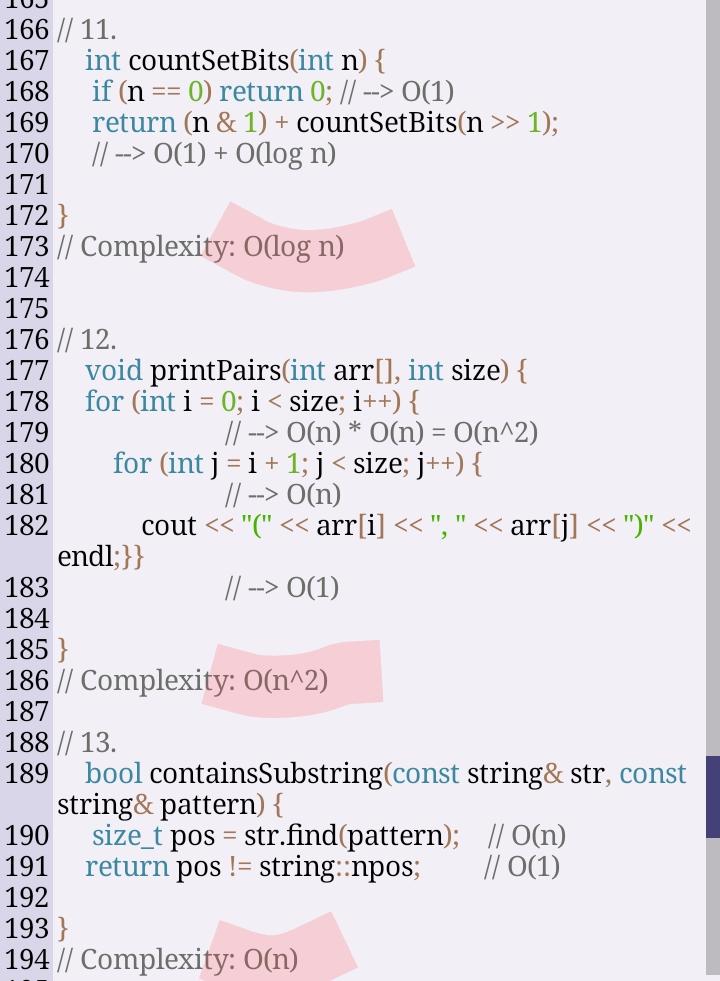


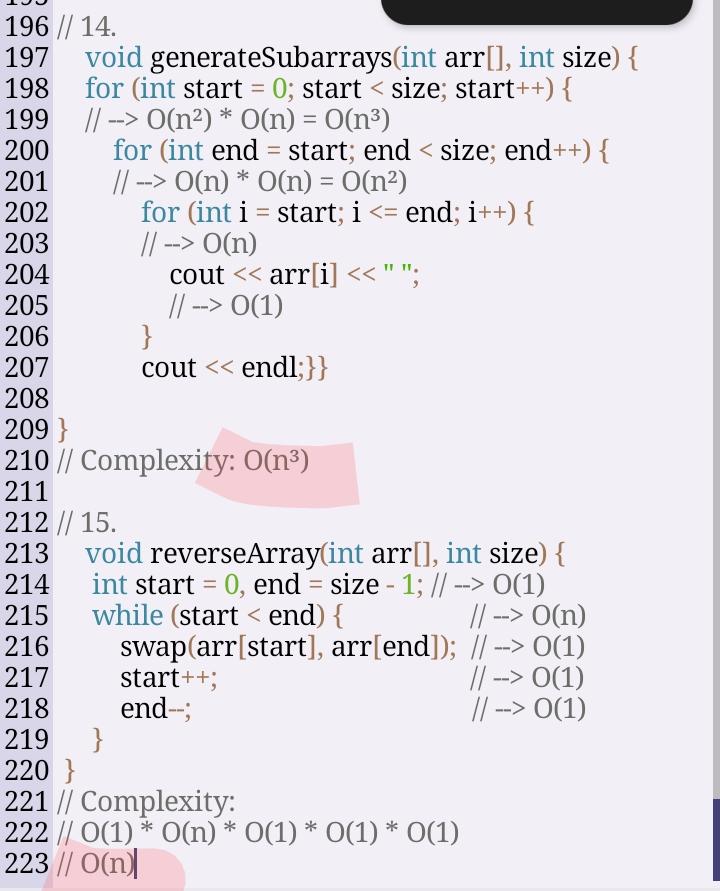












## In raw form

// 1.

Int getFirstElement(int arr[], int size) {

Return arr[0]; // 🡪 O(1)

}

// Complexity: O(1)

// 2.

Int binarySearch(int arr[], int l, int r, int x) {

While (l <= r) {

Int mid = l + (r – l) / 2;

// (n/2)(n/4)

// 🡪 log[sub(2)](n) = O(log n)

If (arr[mid] == x) return mid;

// O(1)

If (arr[mid] < x) l = mid + 1;

// O(1)

Else r = mid – 1;

// O(1)

}

Return -1;

}

// Complexity: O(log n)

// 3.

Int findMax(int arr[], int size) {

Int max = arr[0]; // O(1)

For (int I = 1; I < size; i++) { // O(n)

If (arr[i] > max) max = arr[i]; // O(1)

}

Return max;

}

// Complexity: O(n)

// 4.

Void merge(int arr[], int l, int m, int r) {

// Merge function for merge sort

// Details omitted for brevity

}

Void mergeSort(int arr[], int l, int r) {

If (l < r) {

Int m = l + (r – l) / 2;

// O(1)

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// O(log n)

Merge(arr, l, m, r);

// O(n)

}

}

// Complexity:

// O(log n) \* O(n)

// = O(n log n)

// 5.

Void bubbleSort(int arr[], int size) {

For (int I = 0; I < size – 1; i++) {

// 🡪 O(n^2) \* O(n) = O(n^3)

For (int j = 0; j < size – I – 1; j++) {

// 🡪 O(n) \* O(n) = O(n^2)

If (arr[j] > arr[j + 1]) {

// 🡪 O(n)

Swap(arr[j], arr[j + 1]);

// 🡪 O(1)

}

}

}

}

// Complexity: O(n^3)

// 6.

Void printAllTriplets(int arr[], int size) {

For (int I = 0; I < size; i++) {

// O(n^2) \* O(n) 🡪 O(n^3)

For (int j = I + 1; j < size; j++) {

// O(n) \* O(n) 🡪 O(n^2)

For (int k = j + 1; k < size; k++) {

// O(n)

Cout << arr[i] << “, “ << arr[j] << “, “ << arr[k] << endl;

}

}

}

}

// Complexity: O(n^2)

// 7.

Void permute(string str, int l, int r) {

If (l == r) { // 🡪 O(1)

Cout << str << endl;

} else {

For (int I = l; I <= r; i++) {

// 🡪 O(n)

Swap(str[l], str[i]); // 🡪 O(1)

Permute(str, l + 1, r);

// 🡪 O(n!)

Swap(str[l], str[i]); // Backtrack

// 🡪 O(1)

}

}

}

// Complexity:

// O(1) \* O(n) \* O(n!) \* O(1)

// = O(n \* n!)

// 8.

Int fib(int n) {

If (n <= 1) return n;

// 🡪 O(1)

Return fib(n – 1) + fib(n – 2);

// 🡪 O(n) + O(n) = O(2^n)

}

// Complexity:

// = O(2^n)

// 9.

Void heapify(int arr[], int n, int i) {// Heapify function for heap sort// Details omitted for brevity}

Void heapSort(int arr[], int n) {

For (int I = n / 2 – 1; I >= 0; i--)

// 🡪 O(n)

Heapify(arr, n, i); // 🡪 O(log n)

For (int I = n – 1; I >= 0; i--) {

// 🡪 O(n)

Swap(arr[0], arr[i]); // 🡪 O(1)

Heapify(arr, I, 0); // 🡪 O(log n)

}

}

// Complexity:

// O(n) \* O(log n)

// = O(n log n)

// 10.

Bool hasDuplicate(std::vector<int>& nums){

Std::unordered\_set<int> seen;

For (int num : nums) {

// 🡪 O(n)

If (seen.find(num) != seen.end()) {

// 🡪 O(1)

Return true; // Duplicate found}

Seen.insert(num);}

Return false; // No duplicates

}

}

// Complexity: O(n)

// 11.

Int countSetBits(int n) {

If (n == 0) return 0; // 🡪 O(1)

Return (n & 1) + countSetBits(n >> 1);

// 🡪 O(1) + O(log n)

}

// Complexity: O(log n)

// 12.

Void printPairs(int arr[], int size) {

For (int I = 0; I < size; i++) {

// 🡪 O(n) \* O(n) = O(n^2)

For (int j = I + 1; j < size; j++) {

// 🡪 O(n)

Cout << “(“ << arr[i] << “, “ << arr[j] << “)” << endl;}}

// 🡪 O(1)

}

// Complexity: O(n^2)

// 13.

Bool containsSubstring(const string& str, const string& pattern) {

Size\_t pos = str.find(pattern); // O(n)

Return pos != string::npos; // O(1)

}

// Complexity: O(n)

// 14.

Void generateSubarrays(int arr[], int size) {

For (int start = 0; start < size; start++) {

// 🡪 O(n²) \* O(n) = O(n³)

For (int end = start; end < size; end++) {

// 🡪 O(n) \* O(n) = O(n²)

For (int I = start; I <= end; i++) {

// 🡪 O(n)

Cout << arr[i] << “ “;

// 🡪 O(1)

}

Cout << endl;}}

}

// Complexity: O(n³)

// 15.

Void reverseArray(int arr[], int size) {

Int start = 0, end = size – 1; // 🡪 O(1)

While (start < end) { // 🡪 O(n)

Swap(arr[start], arr[end]); // 🡪 O(1)

Start++; // 🡪 O(1)

End--; // --> O(1)

}

}

// Complexity:

// O(1) \* O(n) \* O(1) \* O(1) \* O(1)

// O(n)