CPSC 1160 – Week 7 Homework

Student: Lana Cambeses

QUESTION 1: Count =, -, >=, --, += operators

(a) Operators count table or list

Operator	Count	Reason
=	2	int t = 0; int i = n - 1
-	1	int i = n - 1
>=	n + 1	i >= 0 (true n times and false once)
--	n	Loop runs n times, and i-- is inside the loop
+=	n	Loop runs n times, and += is inside the loop

(b) Derivation of the total count function T(n)

T(n) = 2 + 1 + n + 1 + n + n

T(n) = 3n + 4

(c) Final Big O complexity and short reasoning

O(n) -> From the T(n), we drop non-dominant terms, as well as constant multiple of the dominant term.

QUESTION 2: Count >=, <, +=, --, ++ operators

(a) Operators count table or list

Operator	Count	Reason
>=	n + 1	for (int i = n - 1; i >= 0; i--) (true n times and false once)
<	(n/2) + 1	for (int j= 0; j < n / 2; j++) (true n/2 times and false once)
+=	n + (n/2)	It happens n times inside the first loop, and then it

		happens n/2 inside the second loop
--	n	It happens n times inside the first loop
++	n/2	It happens n/2 times inside the second loop

(b) Derivation of the total count function T(n)

T(n) = n + 1 + (n/2) + 1 + n + (n/2) + n + (n/2)

T(n) = (3n/2) + 3n + 2

T(n) = (9n/2) + 2

(c) Final Big O complexity and short reasoning

O(n) -> From the T(n), we drop non-dominant terms, as well as constant multiple of the dominant term (and constant divisor too).

QUESTION 3:

(a) Operators count table or list

Ps: listSize = n

Operator	Count	Reason

=	1 + (n – 1) + (n – 1) + (n(n-1)/2) + (n – 1)	int i = 1
		(It happens once at the start of the first loop)
		int currentElement = list[i];
		(This line happens n – 1 times because it's how many times the I loop iterates)
		k = i – 1
		(This line happens n – 1 times because it's how many times the I loop iterates)
		list[k + 1] = list[k];
		(Assuming the worst case and the conditions are always true, this happens I times. But as I keeps on changing at a

		constant rate of 1, we need to use an arithmetic series logic to get the total sum of the iterations. Since the loop goes from 0 to n – 1, the formula is applied as: the sum of each pair 1 + (n – 1) times the amount of pairs that is (n – 1)/2, which is (n(n-1)/2)) list[k + 1] = currentElement; (This happens n - 1 times because its outside of the inner loop, and therefore it runs the same amount of times as the outer loop)
<	n	i < listSize (This line happens n times, true n - 1 times and false once
[]	(n – 1) + (n(n-1)/2 + (2 * (n(n-1)/2)) + n	int currentElement = list[i]; (This line happens n – 1 times, that's how many times the outer loop iterates) list[k] > currentElement (This comparison happens i times, because it's the inner loops sum true times, and when its false, it short circuits) list[k + 1] = list[k]; (This line happens i times; Since the loop goes from 0 to n – 1, the formula is applied as: the sum of each pair 1 + (n – 1) times the amount of pairs that is (n – 1)/2, which is (n(n-1)/2)). As we have two different array accesses, we

		multiply the operation by 2. list[k + 1] = currentElement; (This array access happens n times, which is the number of times the outer loop iterates
>=	(n(n-1)/2 + 1	k >= 0 (This comparison happens i times + 1, because it’s true the inner loops' sum times, and false once)
>	(n(n-1)/2 + 1	list[k] > currentElement (This comparison happens i times + 1, because it’s the inner loops sum true times, and false once, assuming the worst case where the first comparison happens, so we have to check the second)

(b) Derivation of total count function T(n)

$T(n) = 1 + (n – 1) + (n – 1) + (n(n-1)/2) + (n – 1) + n + (n – 1) + (n(n-1)/2 + (2 * (n(n-1)/2)) + n + (n(n-1)/2 + 1 + (n(n-1)/2 + 1$

$T(n) = 1 + n – 1 + n – 1 + (n^2 – n)/2 + n – 1 + n + n – 1 + 3 (n^2 – n)/2 + n + 2(n^2 – n)/2$

$T(n) = n + n – 1 + (n^2 – n)/2 + n – 1 + n + n – 1 + n + 5(n^2 – n)/2$

$T(n) = 6(n^2 – n)/2 + 6n – 3$

$T(n) = 3(n^2 – n) + 6n – 3$

$T(n) = 3n^2 – 3n + 6n – 3$

$T(n) = 3n^2 + 3n – 3$

(c) Final Big O complexity and short reasoning

$O(n^2)$ -> From the T(n), we drop non-dominant terms, as well as constant multiple of the dominant term (and constant divisor too).

QUESTION 4: Count << operators

Operator	Count	Reason
<<	$\log_2(n) + 2 * (\log_2(n) * (2n - 1)) + \log_2(n)$ + $n + 2 * \log_2(n!) + n$	1st loop: cout << "Stirred vial "; (This runs $\log_2(n)$ because we keep on multiplying the value of I by 2 at the end of every iteration.) cout << j << " "; (This runs $2 * (\log_2(n) * 2n - 1)$ because we multiply the number of times the outer loop runs, that's $\log_2(n)$, by how many times the inner loop runs. The inner loop runs i number of times. But as i keeps on changing, we can notice powers of 2 for its values after each iteration, thus forming a geometric series. The geometric series of format $a * (r^{k+1} - 1)/r - 1$ will have 'a' as 1, 'r' as 2, which is the base of the powers for the series, and k, that is the last power, will be $\log_2(n)$. By simplifying everything algebraically, we get $2n - 1$ as a result, and that's what we multiply by the outer loop value. As we have two operators, that's why the multiplication by 2.) cout << endl; (This line runs $\log_2(n)$ times because its also with respect to the outer loop)

		2nd loop: cout << "Cleaned vial residue at "; (This line runs n times, from 1 to n included) cout << q << " "; (This line runs 2 * $\log_2(n!)$ because it goes from p to 0, and q runs $\log_2(p)$ times. But p changes and q is divided by 2 at the end of each iteration, so we sum the logs like $\log_2(1) + \log_2(2) + \log_2(3)\ldots \log_2(n)$ and then, we use the log property to get $\log_2(n!)$. As we have 2 operators inside, then we multiply the expression by two.) cout << endl; (This line happens n number of times, respecting the outer loop)

(b) Derivation of total count function T(n)

$T(n) = \log_2(n) + 2 * (\log_2(n) * (2n - 1)) + \log_2(n) + n + 2 * \log_2(n!) + n$

$T(n) = \log_2(n) + 2 * (2n\log_2(n) - \log_2(n)) + \log_2(n) + n + 2 \log_2(n!) + n$

$T(n) = \log_2(n) + 4n\log_2(n) - 2\log_2(n) + \log_2(n) + n + 2 \log_2(n!) + n$

$T(n) = 2 \log_2(n!) + 4n\log_2(n) + 2n$

(c) Final Big O complexity and short reasoning

$O(n\log_2(n))$-> From the T(n), we have 2 terms with similar orders, and therefore the result depends on the value of n. $n\log_2(n)$ is a bigger function, so drop non-dominant terms, as well as constant multiple of the dominant term.

QUESTION 5: Count =, <=, [], > operators.

Ps: len = n

Operator	Count	Reason

=	3 + len - 1 + 3	int pivot = 0, low = 1, high = len - 1;
		(For this line, we have 1 operator for each assignment, totalizing 3)
		
		int temp = arr[low];
		arr[low] = arr[high];
		arr[high]= temp;
		(For these operators to happen, we consider the worst case, that is; the first two if statements get checked but are false. The loop runs n – 1 times because it stops when high and low get crossed in the middle (minus 1 cause we are not counting the pivot). That would happen n – 1 divided by two, so if we sum them up, we get just n – 1. The thing is that that cannot happen every iteration, because for the place pivot logic, if a swap happens, the next move is to move high and low again. So if we consider that we have those 3 assignments once every 3 iteration (meaning the 3 assignments and then low moves, then high moves), then we would have (3 * (n – 1))/3, thus n – 1, that is a tighter upper bound big O than 3 * (n – 1)
		
		int temp = arr[pivot];
		arr[pivot] = arr[high];
		arr[high] = temp;

		(For these 3 assignments, one operator each, so 3 in total)

<=	len + 2 * (len – 1)	while (low <= high)
		(In the worst case, where we have to make all the comparisons, the loop runs len - 1 times and false once)
		if (arr[low] <= arr[pivot])
		else if(low <= high)
		(For this to be the worst case, the first comparison fails so we can reach the second. Each runs len - 1 times, so that's why 2 * (len – 1).

[]	(len – 1) * (2 + 2 + 4) + 4	if (arr[low] <= arr[pivot])
		else if (arr[high] > arr[pivot])
		int temp = arr[low];
		arr[low] = arr[high];
		arr[high]= temp;
		(For each time the loop runs, that's the length of the array minus 1. Considering that the first two ifs fail and we end up doing the last one, we have 8 array accesses happening).
		int temp = arr[pivot];
		arr[pivot] = arr[high];
		arr[high] = temp;
		(At the end of the function, 4 more array accesses happen)

>	len - 1	else if (arr[high] > arr[pivot]) (Considering that all the first comparisons fail and we go to the second one, the maximum that this line will run is len - 1 times, referring to the

		amount of times the loop runs)

(b) Derivation of total count function T(n)

T(n) = 3 + n - 1 + 3 + n + 2 * (n – 1) + (n – 1) * (2 + 2 + 4) + 4 + n - 1

T(n) = 3 + n - 1 + 3 + n + 2n – 2 + 8n – 8 + 4 + n – 1

T(n) = 13n – 2

(c) Final Big O complexity and short reasoning

O(n)-> From the T(n), we drop non-dominant terms, as well as constant multiple of the dominant term (and constant divisor too).