Esra Eryılmaz
171044046

CSE222 - Spring 2020
Homework 2
03. 13. 2020

PART 1

(1)

	Step	Freq	Total
somefunction (rows = r, cols = c)			
{			
for(i=1; i<=rows; i++) →	2	r+1	2·(r+1)
{			
for(j=1; j<=cols; j++) →	2	r·(c+1)	2·(rc+r)
print(*) →	1	r·c	1·rc
print(newline) →	1	r	1·r
}			+
}			3rc + 5r + 2

The for loop with i as its index will execute r (rows) times. Inside this loop, the for loop with j as its index will execute c (cols) times. Thus the total number of times is r.c times.

Best-case : $O(r \cdot c)$

Worst-case : $O(r \cdot c)$

- It depends on rows and columns.
- Constants can be ignored
- Ignore the low order terms like "5r".

(2) somefunction (a,b)

```
somefunction (a,b)
{
    if (b==0)                           --->
        return 1
    answer = a                          --->
    increment = a                       --->
    for (i=1; i<b; i++)                 --->
    {
        for (j=1; j<a; j++)             --->
        {
            answer += increment         --->
        }
        increment = answer              --->
    }
    return answer                       --->
}
```

Step	Freq	Total
1	1	1
		1
1	1	1
1	1	b
1	b	ab - a
1	a.(b-1)	ab-a-b+1
1	(a-1).(b-1)	b - 1
1	(b - 1)	1
1	1	
		+
		2ab-2a+b+4

- Constants can be ignored.
- Also ignore low order terms.
- If the nested loops contain sizes a and b, the cost is $O(a \cdot b)$

Best-case

If b equals zero then function returns 1.
So in best-case runs only 1 time.

$O(1)$

worst-case

If b is not equal to zero then in worst-case runs 2ab-2a+b+4 times.

$O(a \cdot b)$

③ somefunction (arr[] , arr_len $=n$)

Code	Step	Freq	Total
{			
val = 0	1	1	1
for(i=0 ; i< arr_len/2 ; i++)	2	$\frac{n}{2}+1$	$2\cdot(\frac{n}{2}+1)$
val= val+ arr[i]	3	$n/2$	$3\cdot\frac{n}{2}$
for(i= arr_len/2 ; i<arr_len ; i++)	2	$\frac{n}{2}+1$	$2\cdot(\frac{n}{2}+1)$
val = val - arr[i]	3	$n/2$	$3\cdot\frac{n}{2}$
if(val >=0)	1	1	1
return 1			
else	1	1	1
return -1			+
}			$5n+7$

- If the for loop takes $\frac{n}{2}$ time and i increases or decreases by a constant, the cost is $O(n)$.

- The worst case and the best case scenarios are the same. Because in all cases loops work. ($O(n)$)

Function growth rate is linear

Scanned with CamScanner

4) some function(n)

Code	Step	Freq	Total
{			1
c = 0	1	1	n^2+1
for (i=1 to n*n)	1	n^2+1	n^3+n^2
for (J=1 to n)	1	$n^2.(n+1)$	$n^3.(n^2+2n)$
for (k=1 to 2*J)	1	$n^2.n.\left(\frac{n.(n+1)}{2}.2+n\right)$	n^5+n^4
c = c+1	1	$n^2.n.(n^2+n)$	1
return c	1	1	
}			

$$+\quad 2n^5+3n^4+n^3+2n^2+3$$

when calculating innermost loop:

J	K
1	1 ✓, 2 ✓, 3 ✗
2	1 ✓, 2 ✓, 3 ✓, 4 ✓, 5 ✗
⋮	
n	1 ✓, 2 ✓, ⋮, 2n ✓, 2n+1 ✗

$$(2+4+6+\dots+2n)+\overset{\overset{\text{not executed}}{\uparrow}}{n}$$

$$2.\underbrace{(1+2+3+\dots+n)}+n$$

$$2.\frac{n.(n+1)}{2}+n$$

- If the first loop runs n^2 times and the inner second loop runs n times and the third loop runs n^2+n times, then the cost is $O(n^5)$.

- The worst case and the best case scenarios are the same. Because in all cases all three loops work. $(O(n^5))$

Scanned with CamScanner

5

```
other-function (xp, yp)
{
    temp = xp
    xp = yp          All of them constant    O(1)
    yp = temp
}
```

$arr\_len = n$

```
somefunction (arr[], arr_len)        (arr_len = n)
{
    for(i = 0; i < arr_len - 1; i++)   ) O(n)        (arr_len - 1 = n-1)
    {
        min_idx = i                    ) O(1)

        for(j = i+1 ; j < arr_len ; j++)            (arr_len = n)
  O(1)      if (arr[j] < arr[min_idx])              O(n)           O(n^2)
                min_idx = j

        otherfunction (arr[min_idx], arr[i]) } -> O(1)
    }
}
```

- In somefunction there are loops. The nested loops contain sizes n and n; the cost is $O(n \cdot n) = O(n^2)$.
- Inside that nested loops, there is a function call which cost is constant ($O(1)$). And also there are some constant operations(Also $O(1)$).
- So the total cost is $O(n^2)$. (Quadratic)
- In any cases (Best -worst). The loops are execute. So time complexity stays same. ($O(n^2)$)

⑥ otherfunction (a,b)

```
otherfunction (a,b)
{
    if b == 0;
        return 1                      } O(1)

    answer = a
    increment = a

    for i = 1 to b:  ) O(b)
    {
        for j = 1 to a:
            answer += increment       ) O(a)      O(a·b)

        increment = answer
    }
    return answer
}
```

```
somefunction (arr, arr_len)      n = arr_len
{
    for i=0 to arr_len): ) O(n)
        for j=i to arr_len): ) O(n)
            if otherfunction(arr_len %i , 2) == arr[i]: ) O(1) || O(n)
                print (arr[i])
}
```

$O((n \% i)\cdot 2) = O(2n) = O(n)$

But b is not equal to 0.
b is always 2. So function cannot runs constant time. It will execute always O(n)

Best and worst cases

It will execute always $O(n \cdot n \cdot n) = O(n^3)$ times.

Function growth rate is cubic.

7 otherfunction (x, i)

```
{
    s = 0
    for(j = 1 ; j <= i ; j = j*2)    } O(log_2 i)
        s = s + X[j]
    return s
}
```

arr_len = n

```
somefunction (arr[], arr_len)
{
    for(i=0 ; i <= arr_len - 1 ; i++)    ) O(n)
        A[i] = otherfunction (arr, i) / (i+1)  } O(log_2 i)
    return A
}
```

$O(n \log n)$

- Somefunction first loop runs n times. And function calls another function, which name is otherfunction. It will runs logn times.

The cost is $O(n \log n)$.

Function growth rate is Log-linear.

- Best and worst cases same the total cost. ($O(n \log n)$)

8 somefunction (n)

```
somefunction (n)
{
    res = 0
    J = 1
    if (n<10)
        return n+10                 ) O(1)
    for(i=9 ; i>1 ; i--)            } O(9) -> constant (O(1))
        while( n%i == 0)            ) O(n)
            n = n/i
            res = res + J * i       } O(1)
            J *= 10
    if (n>10)
        return -1                   ) O(1)
    return res
}
```

while loop

inside while loop changing field is n. So loop execution depends on n. Inside that loop there are some operations which costs are also constant. (O(1)) So while loop cost is O(n).

- In somefunction:

Best-case

If n<10, then it returns n+10. So time complexity is constant. O(1).

worst-case

If n is not smaller than 10. It will execute for loop. And time complexity is O(9) . O(n) => (O(9) is constant) we can ignore constants.

Time complexity is O(n).

PART 2

(1) findDistance (Point[ ] , Point P(x1,y1) , arr_len)

```
findDistance ( Point[] , Point P(x1,y1) , arr_len)
{
        newDist = 0
        xDist = 0
        yDist = 0                                           O(1)
        Point(x2,y2) = Point[0]
        xDist = x2 - x1
        yDist = y2 - y1
        oldDist = SquareRoot ( xDist * xDist + yDist * yDist)
                                                            O(n)
        for(i=0 ; i< arr_len ; i++)
        {
                Point(x2,y2) = Point[i]
                xDist = x2 - x1
                yDist = y2 - y1
                newDist = SquareRoot (xDist * xDist + yDist * yDist)
                oldDist = findClosest( newDist , oldDist)   } O(1)
        }
}

find Closest (newDist , oldDist)                            O(1)
{
        if(newDist < oldDist)                   } O(1)
                return newDist
        else                                    } O(1)
                return oldDist
}
```

I sent constants so it will execute constant

(I assume that, I have Point structure)(which includes x and y points)

The total cost is O(n).

- findDistance function time complexity : O(1) + O(n) = O(n) (Linear)

- Best and worst cases are the same with total cost. (O(n))

② findLocalMin(A[], arr_len)

a)

```
}
    localMin = 0
    for( i=0 ; i< arr_len ; i++)
    {
        if( A[i] <= A[i+1]   and   A[i] <= A[i-1])
            localMin = A[i]
    }
}
```

O(1)

O(n)

$\boxed{arr\_len = n}$

findAllMins (A[], arr_len)

b)

```
{
    localMins[]
    k = 0
    for (i=0 ; i< arr_len ; i++)
    {
        if (A[i] <= A[i+1]   and   A[i] <= A[i-1])
        {
            localMins[k] = A[i]
            k++
        }
    }
}
```

O(1)

O(n)

- Time complexities are the same in two functions.
- There is a for loop and inside that loop there are operations in constant time. So the loop contains size n, the cost is in any case (worst-Best) O(n).

3 ques

(3)
```
somefunction ( arr[], arr_len , b)
{
    counter = 0
    for (i=0 ; i< arr_len ; i++)          ) O(n)
        for (j = i+1 ; j < arr_len ; j++)  ) O(n)
            if ( arr[i] + arr[j] == b)     } O(1)
                counter++
    return counter   ) O(1)
}
```

$O(1)$, $O(n)$, $O(n \cdot n) = O(n^2)$

$\boxed{arr\_len = n}$

- Somefunction finds array contains two numbers whose sum is a given number and returns how may numbers that function have.

- If the nested loops contain sizes n and n , the cost is $O(n^2)$. Inside that loop, there is if statement and that statement cost is constant ($O(1)$). so total cost is $\underline{O(n^2)}$

- Best and worst cases, function runs same so time complexities are the same. $\underline{O(n^2)}$

4

$\boxed{arr\_len = n}$

```
isSumChainOfLength ( arr[], arr_len)
{
    for (i = 1 ;   i < arr_len ;  i++)          } O(n)
    {
        if ( somefunction (arr, i, arr[i]) == 0)  } O(n^2)
            return -1
    }
    return 1
}
```

! (somefunction() is from third question)

Best-case

If the function cannot find first number's sum in that array it will return -1. So it will execute (with calling somefunction) $O(n^2)$ times and it will break the for loop. So time complexity

$O(n^2)$

worst case

If the function find all numbers sum in that array, Then for loop executes $O(n)$ and with calling somefunction, it will also executes $O(n^2)$ times.

Total complexity is = $O(n \cdot n^2) = O(n^3)$. (Cubic)