Esra Nur ARICAN 161044028

GIT Department of Computer Engineering CSE 2221505 - Spring 2020

HOMEWORK-2

Part 1

1)

```
some function (rows, cols)
{
  for( i=1; i<=rows; i++)
  {
    for (j=1; j<=cols; j++)
      print(*)
    print(newline)
  }
}
```

steps/exec	freq	total
1	rows	rows
1	rows. cols	rows. cols
1	rows cols	rows. cols
1	rows	rows
		2rows.cols + 2rows

Table method used in this question. worst case, best case, average cases are the same as : $\Theta(rows.cols)$ (or we can say $O(row.cols)$)

2)

```
some function (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
}
```

steps/exec	freq	total
1	1	1
1	1	1
1	1	1
1	1	1
1	b	b
1	ab	ab
1	ab	ab
1	b	b
1	1	1
		2ab+2b+ 5

Total is $2ab + 2b + 5$

worst case is $O(2ab + 2b + 5)$ → ignore lower order terms = $\Theta_{worst}(ab)$

Best case if $b == 0$ situation → $\Theta_{best}(1)$

Average is ; $P(T)\ T_1 + P(F)T_2$ if we use this formula:

$$\frac{1}{2}\Theta(1) + \frac{1}{2}\Theta(ab) = \frac{\Theta(ab)}{2} = \Theta_{av.}(ab)$$

3)

```
Some function (arr [ ], arr_len) {
    val = 0
    for(i=0 ; i< arr_len/2 ; i++)
        val = val + arr [i]
    For (i=arr_len/2 ;  I<arr_len ; i++)
        val = val - arr [i]
    if (val >= 0)
      return 1
    else
      return -1
}
```

steps/exec	freq	total
1	1	1
1	arr_len/2	arr_len/2
1	arr_len/2	arr_len/2
1	arr_len/2	arr_len/2
1	arr_len/2	arr_len/2
1	1	1
1	1	1
1	1	1
1	1	1
		$4\frac{arr\_len}{2} + 5$

Best, worst, average case are not significant. All same. $= 2arr\_len + 5$

T_{best} $\Theta(arr\_len)$

T_{worst} $O(arr\_len)$

T_{av} $\Theta(arr\_len)$

4)

```
somefunction (n) {
    c=0
    for(i=1 to n*n)
      for(j=1 to n)
        for(k=1 to 2*j)
          c= c+1
    return c
}
```

step/exec	freq	total
1	1	1
1	n^2	n^2
1	n^3	n^3
1	$n^3 \cdot 2j$	$2n^3 j$
1	$n^3 \cdot 2j$	$2n^3 j$
1	1	1
		$4n^3 j + n^3 + n^2 + 2$

For table method : $\Theta(4n^3 j + n^3 + n^2 + 2) = \Theta(4n^3 j) = \Theta(n^3 j)$

Best, worst, average cases are same.

5)

```
other function(xp, yp) {
    temp = xp        } Q(1)
    xp = yp          } Q(1)
    yp = temp        } Q(1)
}                                   } Q(1)

some function(arr[], arr_len) {
    for(i=0; i<arr_len-1; i++) {
        min_idx = i                 } Q(1)
        for(j=i+1; j<arr_len; j++)  } Q(n^2)
            if(arr[j] < arr[min_idx])
                min_idx = j         } Q(1)
        other function(arr[min_idx], arr[i]) } Q(1)
    }
}
```

Q(n-1) = Q(n)

* Let's say arr_len = n
* outer for loop Q(n-1)
* to calculate inner loop I used this;

	inner loop	
i = 0	1 to n	= n-1
i = 1	2 to n	= n-2
i = 2	3 to n	= n-3
⋮	⋮	⋮
i = n-1	n to n	= n-n

$n \cdot n + \frac{n \cdot n-1}{2}$

$n^2 + \frac{n^2-n}{2}$

I calculated how many times inner loop executed.

Best case = $\Theta(n^2)$
worst case = $O(n^2)$
average = $\Theta(n^2)$

- some function calls other function inside. But other function's complexity is Θ(1) while somefunction's $\Theta(n^2)$ so in general we take $\Theta(n^2)$ //

6)

```
other funct(a,b) {
    If(b==0)  } Q(1)
      return 1  } Q(1)
    answer = a  } Q(1)
    increment = a } Q(1)
    for i=1 to b:
    {
      for j=1 to b:
        answer += increment  } Q(ab)
      increment = answer     } Q(b)
    }
    return answer  } Q(1)
}
```

For T1; if b==0 → Best case $\underline{\Theta(1) = \Omega(1)}$
if b!= → worst case $\underline{O(ab) = \Theta(ab)}$

$T_{av} = P(T) \cdot T1 + P(F) \cdot T2$

$\frac{1}{2} \cdot \Theta(1) + \frac{1}{2} \Theta(ab) = \frac{1}{2} \Theta(ab) = \underline{\Theta(ab)}$

continued →

continued next page.

6 continued)

```
somefunction(arr, arr_len){
    for(i=0 to arr_len):
        for(j=i to arr_len):
            if(otherfunction(arr[i], 2)== arr[j]:
                print(arr[i], arr[j])
```

Changed ~~if otherfunction(arr[j],2) == arr[j]~~
~~print(arr[j], arr[i])~~

```
}
```

let's say arr_len = n.

outer	inner	
0	0 to n	n
1	1 to n	n-1
2	2 to n	n-2
⋮	⋮	⋮
n	n to n	n-n

$$\frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

so $\Theta(\frac{n^2+n}{2}) = \Theta(n^2)$

for loop = $\Theta(arr\_len^2)$
also it calls other function with $\Theta(ab)$
result is $\Theta(arr\_len^2) \cdot \Theta(ab) = \Theta(arr\_len^2 \cdot a \cdot b)$

7)
```
otherfunction(x,j){
    s=0                                  } Θ(1)
    for(j=1; j<=i; j=j*2)                } Θ(log2(i))   } Θ(log2(i))
        s=s+x[i]                         } Θ(1)
    return s                             } Θ(1)
}
somefunction(arr[], arr_len){
    for(i=0; i<=arr_len-1; i++)
        A[i] = otherfunction(arr, i)/(i+1) } Θ(log2(i))  } Θ(arr_len)   } Θ(arr_len · log2(i))
    return A                             } Θ(1)
}
```

8)
```
somefunction(n){
    res = 0                              } Θ(1)
    j=1                                  } Θ(1)
    if(n<10)                             } Θ(1)
        return n+10
    for(i=9; i>1; i--)                   } Θ(1)  // repeated 9 times for sure.
        while(n%i==0)                    } √n (?) not sure?
            n=n/i
            res=res+j*i
            j*=10
    if(n>10)
        return -1
    return res
}
```

Best case if
n<10 = O(1) = Θ(1)

Part 2

1)

```
findClosest ( arr[][], x, y ){
    int temp, distance;
    distance = sqrt((arr[0][0]- x) * (arr[0][0]-x) + (arr[0][1]-y)(arr[0][1]-y));
    for ( i=0  to  i< arr.size() ; i++){
        temp=0
        temp= sqrt(( arr[i][0]-x)*(arr[i][0]-x)+(arr[i][1]-y)*(arr[i][1]-y))
        if (temp < distance)
            distance = temp;
    }
    return  distance
}
```

First I found the distance of first element of array to the given point. Then I assign this value to "distance" variable. Later, in a for loop I found every index's distance between given point. I checked this temp value and distance and if it smaller, assing it to the distance value. Finally return the distance.

Complexicty of this pseudocode is $O(n)$ (n = arr.size())

2) a)

```
findLocalMinimum (arr [], int begin, int end, n){
    middle = (begin + (end - begin)) / 2;
    if( (middle== 0 || arr[middle-1] > arr[middle] ) and
        middle== n-1  || arr[middle] < arr[middle+1])
        return middle;
    if(middle > 0  and  arr[middle-1] < arr[middle])
        return  findLocalMinumum (arr, begin, (middle-1), n);
    else
        return findLocalMinimum( arr, (middle+1), end, n);
}
```

I used binary search for this problem. I found middle element and checked if it is local minimum, if middle element is not local minimum, then I called the function for left half. else I called function right half. Time complexity of this function is $O(\log n)$. Because we know that the complexity of recursive calls are $O(\log n)$.

2) b)

```
findAllLocalMin ( arr [ ], size) {
    resultArr [size];
    for ( int i = 1  to   size-1, i++)
        int j = 0;
        if ( arr [i] > arr [i-1]  and  arr [i] < arr [i+1] ) {
                print (arr [i]);
                resultArr [j] = arr [i];
                 j++;
        }
        return  resultArr;
}
```

O(n) - one for loop, other calculations are constant time.

Starting from 1. index of array and looking every index's left and right neighborhoods if that element in that index is local minimum, assign it to the new resultArr and print it. At the end return that array which holds all local minimums.

Time complexity of this function is O(n) (n= size of array.)

3)

```
isContainSum (arr [ ], size, number) {
    i=0 , j=0 ;
    for ( i=0   to   i < size ; i++ ) {
       for ( j = 0  to   j< size ;  j ++) {
           if (  i not equal j )
              if (arr [i] + arr [j] == number)
                 return  true
       }
    }
    return  false
}
```

Starting from the first index, calculating every index's sum with each other and we find the given return true, otherwise we return false.

There is two for loop as nested. And time complexity of this function is $O(n^2)$ (n= size of the array). This is the worst case : $O(n^2)$

Esra Nur ARKAN 161044028

4)

```
Is sumchain ( arr[], size )
    i=0, j=0 , m=0;
    for( i = 1;  i< size; i++){
      for( j=0;  j<size ; j++){
          if (arr[i] + arr[j] == arr[m] }
                print( arr[j] , arr[i], arr[m])
          }
          m+1
      }
    }
}
```

O(n) O(n) O(n²)

I used almost same algoritm in this question except checking i not equal j. Because { 1, 2, 3, 5, 10, 13 ... } "i not equal j" statement wouldn't let me find 5+5 = 10 , so I removed this statement, and checked every index of the array is a sum chain. The complexity of this function is O(n²) (n= size). Because we have nested for loops and each one take O(n) time.