ANALYSIS OF ALGORITHMS

Chapter2/Part1

Prepared by: Enas Abu Samra

KEY POINTS OF CHAPTER2

- Analysis of Algorithms.
- Calculating the Running Time of a program.
- Order of Growth.
- Best Case, Average Case, Worst Case.
- Analyzing the Time Efficiency of non-recursive Algorithms.
- Analyzing the Time Efficiency of recursive Algorithms.

EFFICIENCY OF ALGORITHMS

- **Efficiency:** number of resources used by an Algorithm.
- An algorithm is considered efficient if its resource consumption is below some acceptable level. It should run in a reasonable amount of time on an available computer or hardware specifications.
- The two most common used measures are:
  - ✓ **Time efficiency (time complexity):** How long does the algorithm take to complete/ how fast an algorithm runs.
  - ✓ **Space efficiency (space complexity):** How much working memory is needed by the algorithm/ refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

EFFICIENCY OF ALGORITHMS

- For an algorithm to be efficient, the above two factors should be optimized, i.e., the algorithm should fulfill all the proper requirements with **minimum amount of time** and within **specified space limits should not overload the memory.**

- The term "analysis of algorithms" is usually used in a narrower, technical sense to mean an investigation of an algorithm's efficiency with respect to two resources: running time and memory space.

SPACE EFFICIENCY

- For large quantities of data space/memory used should be analyzed.
- The major components of space are:
  - ✓ **Instruction space →** the amount of memory needed by the code.
  - ✓ **Data space →** the amount of memory needed for the data on which the code operates.
  - ✓ **Run-time stack space.**

EXAMPLES/SPACE COMPLEXITY

```
Algorithm Swap(a,b)
{       temp=a;  ------- 1
         a=b;       -------- 1
         b=temp;   -------- 1
}    ---------------------------------
        f(n) = 3
```

space

a

b

temp

----

s(n)= 3

EXAMPLES/SPACE COMPLEXITY

```
Algorithm  sum(A,n)
{     s=0;
       for(i=0;i<n;i++)
        {
            s=s+A[i];
        }
Return s;

}
```

space

A=n

n=1

i=1

s=1

---

s(n)= n+3

EXAMPLES/SPACE COMPLEXITY

```
Algorithm  sum(A,B,n)
{          for(i=0;i<n;i++)
           {
          for(j=0;j<n;j++)
            { c[i,j]=0;
              for(j=0;j<n;j++) {
          C[i,j]=A[i,j]+B[i,j];
                }
             }
           }
}
```

space:

A	n^2
B	n^2
c	n^2
n	1
j	1
I	1

$3\,n^2 + 3$

TIME EFFICIENCY

- Running time depends upon:
  - ✓ Compiler used
  - ✓ R/W speed to Memory and Disk
  - ✓ Machine Architecture: 32-bit vs 64
  - ✓ Input size (rate of growth of time)

ANALYSIS OF TIME COMPLEXITY

- When analyzing for time complexity we can take **two approaches**:

1. Estimation of running time – Frequency Count Method (FCM)

Running time of an algorithm is the number of FCM instructions it executes By analysis of the code, we can do:

- ✓ **Operation counts -** select operation(s) that are executed most frequently and determine how many times each is done.
- ✓ **Step counts -** determine the total number of steps, possibly lines of code, executed by the program.

2. Order of magnitude/asymptotic categorization - gives a general idea of performance.

COMPARING ALGORITHMS

Question: Given two algorithms A and B, how do we know which is faster?

Answer: Implement and run both and compare the time each takes!

To compare two algorithms, we can implement them, run them and compare their running times.

CHALLENGES

- **The running time of a program is hardware and software dependent.**

  We need to run both algorithms on the same machine (or on machines with the same specs), using the same programming language, the same compiler, etc.

- **The running time of a program depends on the input size and on the input type.**

  We need to run the programs as many times as needed to cover all possible input sizes and types that might affect the behavior of the programs.

- **Running the programs might take a long time!**

  Takes as long as the fastest of the two programs requires.

RULES OF FREQUENCY COUNT METHOD

- **For comments and declarations the step count is 0.**
  - ✓ Comments are not executed, and declarations are not needed while writing the algorithm.
- **For the return, assignment, arithmetic's, and logic statements the step count is 1.**
  - ✓ Assignment statement is simply assigning a left hand value to a right hand value, while the return statement is just returning a value, Arithmetic's statement (e.g., +, -, *, /, …), and Logic statement (e.g., >, <, …).
- **Only consider higher order exponents.**
  - ✓ E.g.; $3n^2 + 4n$. We will only consider $3n^2$ as it has the higher order exponent.
- **Ignore the constant value multipliers.**
  - ✓ We are left with $3n^2$, our constant multiplier is 3, we ignore it and are left with n^2. Our time complexity is $O(n^2)$.

RULES OF FREQUENCY COUNT METHOD

- **Loops → Summation (see Math sheet)**

$$\sum_{\boldsymbol{i=L}}^{\boldsymbol{U}} \boldsymbol{C} = \sum_{i=L}^{U} C(\mathrm{U} - \mathrm{L} + 1)$$

U: upper, L: lower, C: constant, i: counter.

- **For loop: C→ 2 steps** (comparison and counter update)
- **While loop: C → 1 step** (comparison).
- **Statements inside loops** (C → **how many statements** and the upper mins one because last iteration is not included).

RULES OF FREQUENCY COUNT METHOD

- **For loop:**
- $\sum_{\boldsymbol{i=L}}^{\boldsymbol{U}} \mathbf{2} = \sum_{i=L}^{U} 2(\mathrm{U} - \mathrm{L} + 1)$ U+1 if there is an equal operator in the condition part.

- **While loop:**
- $\sum_{\boldsymbol{i=L}}^{\boldsymbol{U}} \mathbf{1} = \sum_{i=L}^{U} 1(\mathrm{U} - \mathrm{L} + 1)$ U+1 if there is an equal operator in the condition part.

- **Statements inside loops:**
- $\sum_{\boldsymbol{i=L}}^{\boldsymbol{U-1}} \boldsymbol{C} = \sum_{i=L}^{U-1} C(\mathrm{U} - 1 - \mathrm{L} + 1)$ C: number of statements

ORDER OF RANKS

order of growth

	name	function
	constant	1
good	logarithmic	$\log(n)$
good		$\sqrt{n}$
fine	linear	n
fine	linearithmic	$n \log(n)$
fine		$n\sqrt{n}$
bad	quadratic	n^2
bad	cubic	n^3
horrible	exponential	2^n
horrible	exponential	3^n
horrible	factorial	$n!$

EXAMPLES/TIME COMPLEXITY

Example1:

//assume n =2

```
For (int i= 1; i <= n; i++)
  display(i)
```

Tracing:

i	**i<=2**	**print sta.**
1	1<=2	Done (iteration1 ➔ compare and increase counter)
2	2<=2	Done (iteration2 ➔ compare and increase counter)
3	3<=2	----- (last iteration ➔ just compare)

Number of operations the loop is executed when n = 2→ 6

Number of times the print statement is executed when n = 2→ 2

If n = 3, operations of loop will be (8) and print statement will be executed (3) times.

If n = 4, operations of loop will be (10) and print statement will be executed (4) times.

EXAMPLES/TIME COMPLEXITY

Number of Steps using Random Access Method:

For (int i= 1; i <= n; i++) → $\sum_{i=1}^{n+1} \mathbf{2} = \sum_{i=1}^{n+1} 2(n + 1 - 1 + 1) = 2n + 2$

display(i) → $\sum_{i=1}^{n} \mathbf{1} = \sum_{i=1}^{n} 1(n - 1 + 1) = n$

f(n) = 2n + 2 + n = 3n + 2, **f(n) = O(n)**

EXAMPLES/TIME COMPLEXITY

Example2:

//assume n =2

```
For (int i= 1; i <n; i++)
  display(i)
```

Number of Steps using Random Access Method:

For (int i= 1; i < n; i++) → $\sum_{i=1}^{n} \mathbf{2} = \sum_{i=1}^{n} 2(n - 1 + 1) = 2n$

display(i) → $\sum_{i=1}^{n-1} \mathbf{1} = \sum_{i=1}^{n-1} 1(n - 1 - 1 + 1) = n - 1$

$f(n) = 2n + n - 1 = 3n - 1$, $\mathbf{f(n) = O(n)}$

If n = 2, operations of loop will be (4) and print statement will be executed one time.

If n = 3, operations of loop will be (6) and print statement will be executed (2) times.

EXAMPLES/TIME COMPLEXITY

Example3:

//assume n =2

int i =1; → 1

while (i < = n) → $\sum_{i=1}^{n+1} \mathbf{1} = \sum_{i=1}^{n+1} 1(n+1-1+1) = n+1$

{ display(i)

i ++ } → $\sum_{i=1}^{n} \mathbf{2} = \sum_{i=1}^{n} 2(n-1+1) = 2n$

f(n) = 1 + n + 1 + 2n = 3n + 2 , **f(n) = O(n)**

If n = 2, operations of loop will be (3) and loop statements will be executed (4) times.

EXAMPLES/TIME COMPLEXITY

Example4:

//assume n =2

int i =1; → 1

while (i < n) → $\sum_{i=1}^{n} \mathbf{1} = \sum_{i=1}^{n} 1(n-1+1) = n$

{ display(i)

i ++ } → $\sum_{i=1}^{n-1} \mathbf{2} = \sum_{i=1}^{n-1} 2(n-1-1+1) = 2n - 2$

f(n) = 1 + n + 2n - 2 = 3n - 1 , **f(n) = O(n)**

If n = 2, operations of loop will be (2) and loop statements will be executed (2) times.

EXAMPLES/TIME COMPLEXITY

Example5:

For (int i= 2; i <= n; i++) → $\sum_{\boldsymbol{i=2}}^{\boldsymbol{n+1}} \mathbf{2} = \sum_{i=2}^{n+1} 2(\mathrm{n} + 1 - 2 + 1) = 2\mathrm{n}$

display(“hi”) → $\sum_{\boldsymbol{i=2}}^{\boldsymbol{n}} \mathbf{1} = \sum_{i=2}^{n} 1(\mathrm{n} - 2 + 1) = \mathrm{n} - 1$

f(n) = 2n + n -1 = 3n - 1, **f(n) = O(n)**

EXAMPLES/TIME COMPLEXITY

Example6:

int sum = 0; → 1

For (int i= 2; i < n^2 – 1; i++) → $\sum_{i=2}^{n2-1} \mathbf{2} = \sum_{i=2}^{n2-1} 2(n2 - 1 - 2 + 1)$ = $2n^2$ -4

$\left.\begin{array}{l}\text{\{ sum += i;} \\ \text{display(sum) ;\}}\end{array}\right\}$ → $\sum_{i=2}^{n2-2} \mathbf{2} = \sum_{i=2}^{n2-2} 2(n2 - 2 - 2 + 1)$ = $2n^2$ - 6

f(n) = 1 + $2n^2$ - 4 + $2n^2$ – 6 = $4n^2$ - 9, **f(n) = O(n^2)**

EXAMPLES/TIME COMPLEXITY

Example7:

int sum = 0; → 1

For (int i= n; i <= 2n; i++) → $\sum_{i=n}^{2n+1} \mathbf{2} = \sum_{i=n}^{2n+1} 2(2n + 1 - n + 1) = 2n + 4$

sum += 2; → $\sum_{i=n}^{2n} \mathbf{1} = \sum_{i=1}^{n} 1(2n - n + 1) = n + 1$

f(n) = 1 + 2n + 4 + n + 1 = 3n + 6, **f(n) = O(n)**

EXAMPLES/TIME COMPLEXITY

Example8:

For (int i= n; i > =1; i--) ➔ For (int i= 1; i <= n; i++) → 2n +2

display("hi") → n

f(n) = 2n + 2 + n = 3n + 2, **f(n) = O(n)**

EXAMPLES/TIME COMPLEXITY

Example9:

```
For (int i= 1; i <= n; i += 2)
  display("hi")
```

Note:

iteration(i) ➔	0	1	2	3	….	n		
counter(j) ➔	1	3	5	7	….	t	➔	2(i) + 1

2(i) + 1 ➔ **i = (n-1)/2**

EXAMPLES/TIME COMPLEXITY

```
For (int i= 0; i <= (n-1)/2; i++)
  display("hi")
```

Complexity:

for loop ➔

$$\sum_{i=0}^{(n-1)/2+1} \mathbf{2} = \sum_{i=0}^{(n-1)/2+1} 2((n-1)/2+1-0+1) = 2((n\text{-}1)/2 + 2)$$

statement inside loop ➔

$$\sum_{i=0}^{(n-1)/2} \mathbf{1} = \sum_{i=0}^{(n-1)/2} 1((n-1)/2-0+1) = (n\text{-}1)/2 + 1$$

f(n) = 2((n-1)/2 + 2) + (n-1)/2 + 1 ,   **f(n) = O(n)**

EXAMPLES/TIME COMPLEXITY

Example10:

```
For (int i= 2; i <= n; i += 3)
  display("hi")
```

Note:

iteration(i) ➔ 0 1 2 3 …. n

counter(j) ➔ 2 5 8 11 …. t ➔ 3(i) + 2

3(i) + 2 ➔ **i = (n-2)/3**

EXAMPLES/TIME COMPLEXITY

For (int i= 0; i <= (n-2)/3; i++)

display(“hi”)

Complexity:

for loop ➔

$\sum_{\boldsymbol{i=0}}^{\boldsymbol{(n-2)/3+1}} \mathbf{2} = \sum_{i=0}^{(n-2)/3+1} 2((n-2)/3+1-0+1)$ = 2((n-2)/3 + 2)

statement inside loop ➔

$\sum_{\boldsymbol{i=0}}^{\boldsymbol{(n-2)/3}} \mathbf{1} = \sum_{i=0}^{(n-2)/3} 1((n-2)/3-0+1)$ = (n-2)/3 + 1

f(n) = 2((n-2)/3 + 2) + (n-2)/3 + 1 , **f(n) = O(n)**

RULE(1)

#Rule 1:

```
for(int i =  a;  i<= function;  i += b)  // or minus
any statement
```

Notes:

a, b ➔ any number, function ➔ constant/n/n^2/ ….

Complexity ➔ the higher order limit in function.

Review the previous examples.

EXAMPLES/TIME COMPLEXITY

Example11:

```
For (int i= 1; i <= n; i*=2)
  display("hi")
```

Note:

iteration(i) ➔ 0 1 2 3 …. n

counter(j) ➔ 1 2 4 8 …. t ➔ $2^i * 1 = 2^i$

2^i ➔ **$i = \log_2 n$**

EXAMPLES/TIME COMPLEXITY

For (int i= 0; i <= $\log_2 n$; i++)

display("hi")

Complexity:

for loop ➔

$$\sum_{i=0}^{log2n+1} \mathbf{2} = \sum_{i=0}^{log2n+1} 2(\log 2n + 1 - 0 + 1) = 2(\log_2 n + 2) = 2\log_2 n + 4$$

statement inside loop ➔

$$\sum_{i=0}^{log2n} \mathbf{1} = \sum_{i=0}^{log2n} 1(\log 2n - 0 + 1) = \log_2 n + 1$$

$f(n) = 2\log_2 n + 4 + \log_2 n + 1 = 3\log_2 n + 5$, **$f(n) = O(\log_2 n)$**

EXAMPLES/TIME COMPLEXITY

Example12:

```
For (int i= n; i > 0 ; i= i/2)          →     For (int i= 1; i <= n; i*=2)
  display("hi")
```

Exactly as the previous.

RULE(2)

#Rule 2:

```
for(int i =  a;  i<= function;  i *= b)  // or division
  any statement
```

Notes:

a, b → any number, function → constant/n/n2/ ….

Complexity → $\log_b$(the higher order limit in function)

Review the previous examples.

EXAMPLES/TIME COMPLEXITY

Example13:

```
For (int i= 0; i <= n; i++)          → O(n)
 For (int j= 0; j <= n; j++)         → O(n)
   display("hi")
```

$$f(n) = O(n * n) = O(n^2)$$

EXAMPLES/TIME COMPLEXITY

Example14:

For (int i= 1; i <n; i++) → O(n)

For (int j= 0; j <= n; j++) → O(n)

display("hi")

f(n) = O(n * n) = O(n^2)

EXAMPLES/TIME COMPLEXITY

Example15:

For (int i= 1; i <3n; i++) → O(n)

For (int j= 1; j <= n; j++) → O(n)

display("hi")

$f(n) = O(n * n) = O(n^2)$

EXAMPLES/TIME COMPLEXITY

Example16:

For (int i= 1; i <= n; i *= 2) → $O(\log_2 n)$

For (int j= 1; j <= n; j++) → $O(n)$

display("hi")

$$\mathbf{f(n) = O(n * \log_2 n) = O(n\log_2 n)}$$

EXAMPLES/TIME COMPLEXITY

Example17:

For (int i= 1; i <= n; i++) $\rightarrow O(n)$

For (int j= 1; j <= n; j*= 2) $\rightarrow O(\log_2 n)$

display("hi")

$$\mathbf{f(n) = O(n * \log_2 n) = O(n\log_2 n)}$$

EXAMPLES/TIME COMPLEXITY

Example18: (important)

```
For (int i= 0; i <3; i++)              → O(3)
 For (int j= 0; j <= n; j++)            → O(n) → O(n)
   display("hi")
```

$$f(n) = O(3 * n) = O(n)$$

EXAMPLES/TIME COMPLEXITY

Example19:

For (int i= n; i >0; i = i - c) → O(n)

For (int j= i + 1; j <= n; j+= c) → O(n)

display("hi")

$f(n) = O(n * n) = O(n^2)$

EXAMPLES/TIME COMPLEXITY

Example20:

```
For (int i= 0; i <= n; i++)          → O(n)
    display("hi")
For (int j= 0; j <= n; j++)          → O(n)
    display("hi")
```

f(n) = O(n + n) = O(2n) = O(n)

EXAMPLES/TIME COMPLEXITY

Example21:

```
For (int i= 0; i <n; i++)              → O(n)
 {For (int j= 0; j <= n; j++)          → O(n)
     display("hi")

 For (int j= 0; i <= n; j++)           → O(n)
     display("hi")
 }
```

$f(n) = O(n * (n+ n)) = O(n * 2n) = O(2n^2) = O(n^2)$

EXAMPLES/TIME COMPLEXITY

Example22:

```
For (int i= 0; i <n; i++)            → O(n)
 For (int j= 0; j <n; j += 2)        → O(n)
  For (int k= 10; k <n; k++)         → O(n)
     display("hi")
```

$f(n) = O(n * n * n) = O(n^3)$

EXAMPLES/TIME COMPLEXITY

Example23:

```
For (int i= 0; i <n²; i++)            →  O(n²)
   display("hi")

 For (int i= 0; i <n; i  += 2)         → O(n)
   For (int j= 0; j <n; j += 2)        → O(n)
      display("hi")
```

$f(n) = O(n^2 + (n * n)) = O(2n^2) = O(n^2)$

EXAMPLES/TIME COMPLEXITY

Example24: (special case, not included in material)

```
For (int i= 1; i <=n; i ++)
   For (int j= 1; j <= i; j++)
      display("hi")
```

$f(n) = O(n^2)$

END OF CHAPTER2/PART1