Analysis and Design of Algorithms
Lecture 2

Analysis of Algorithms I

Dr. Mohamed Loey
Lecturer, Faculty of Computers and Information
Benha University
Egypt

Table of Contents

Analysis of Algorithms

- **Analysis of Algorithms** is the determination of the amount of **time**, **storage** and/or other **resources** necessary to execute them.
- Analyzing algorithms is called **Asymptotic Analysis**
- **Asymptotic Analysis** evaluate the performance of an algorithm

Time complexity

Time complexity

- **time complexity** of an algorithm quantifies the amount of **time** taken by an algorithm

- We can have three cases to analyze an algorithm:
  1) Worst Case
  2) Average Case
  3) Best Case

Time complexity

- ❑ Assume the below algorithm using **Python** code:

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- **Worst Case Analysis:** In the worst case analysis, we calculate upper bound on running time of an algorithm.

```python
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

Time complexity

- **Worst Case Analysis:** the case that causes maximum number of operations to be executed.
- For Linear Search, the worst case happens when the element to be searched is not present in the array. (example : search for number 8)

2	**3**	**5**	**4**	**1**	**7**	**6**

Time complexity

- **Worst Case Analysis:** When x is not present, the search() functions compares it with all the elements of **arr** one by one.

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

Time complexity

- The worst case time complexity of linear search would be O(n).

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

Time complexity

- ❑ **Average Case Analysis:** we take all possible inputs and calculate computing time for all of the inputs.

```python
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- **Best Case Analysis:** calculate lower bound on running time of an algorithm.

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

Time complexity

- The best case time complexity of linear search would be O(1).

```python
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

Time complexity

- **Best Case Analysis:** the case that causes minimum number of operations to be executed.
- For Linear Search, the best case occurs when x is present at the first location. (example : search for number 2)
- So time complexity in the best case would be $\Theta(1)$

2	**3**	**5**	**4**	**1**	**7**	**6**

Time complexity

- Most of the times, we do **worst case** analysis to analyze algorithms.

- The **average case** analysis is not easy to do in most of the practical cases and it is rarely done.

- The **Best case** analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information.

Asymptotic Notations

1) **Big O Notation: is** an Asymptotic Notation for the worst case.

2) **Ω Notation** (omega notation)**: is** an Asymptotic Notation for the best case.

3) **Θ Notation** (theta notation) : **is** an Asymptotic Notation for the worst case and the best case.

Big O Notation

1) O(1)

- Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function. For example swap() function has O(1) time complexity.

```
def swap(s1, s2):
    return s2, s1
```

Big O Notation

- A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
# c is constant
c=4
for i in range(1,c):
    #some O(1) expressions
    #print(i)
```

2) O(n)

- Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example the following loop statements have O(n) time complexity.

Big O Notation

2) O(n)

```
# n is variable
# c is increment
for i in range(1,n,c):
    #some O(1) expressions
    print(i)
```

2) O(n)

❑ Another Example:

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

Big O Notation

3) $O(n^c)$

- Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following loop statements have $O(n^2)$ time complexity

3) $O(n^2)$

```python
# n is variable
# c is increment
for i in range(1,n,c):
    #some O(1) expressions
    for j in range(1,n,c):
        #some O(1) expressions
        print(i,j)
```

❑ Another Example

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
        // some O(1) expressions
    }
}

for (int i = n; i > 0; i += c) {
    for (int j = i+1; j <=n; j += c) {
        // some O(1) expressions
}
```

4) O(Logn)

- Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
# n is variable
# c is constant
i=2
while i<=n:
    print(i)
    i=i*c
```

4) O(Logn)

➢ Another Example

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

5) O(LogLogn)

- Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant.

```
# n is variable
# c is constant
i=2
while i<=n:
    print(i)
    i=i**c
```

5) O(LogLogn)

➢ Another Example

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {
    // some O(1) expressions
}
```

- ❑ How to combine time complexities of consecutive loops?

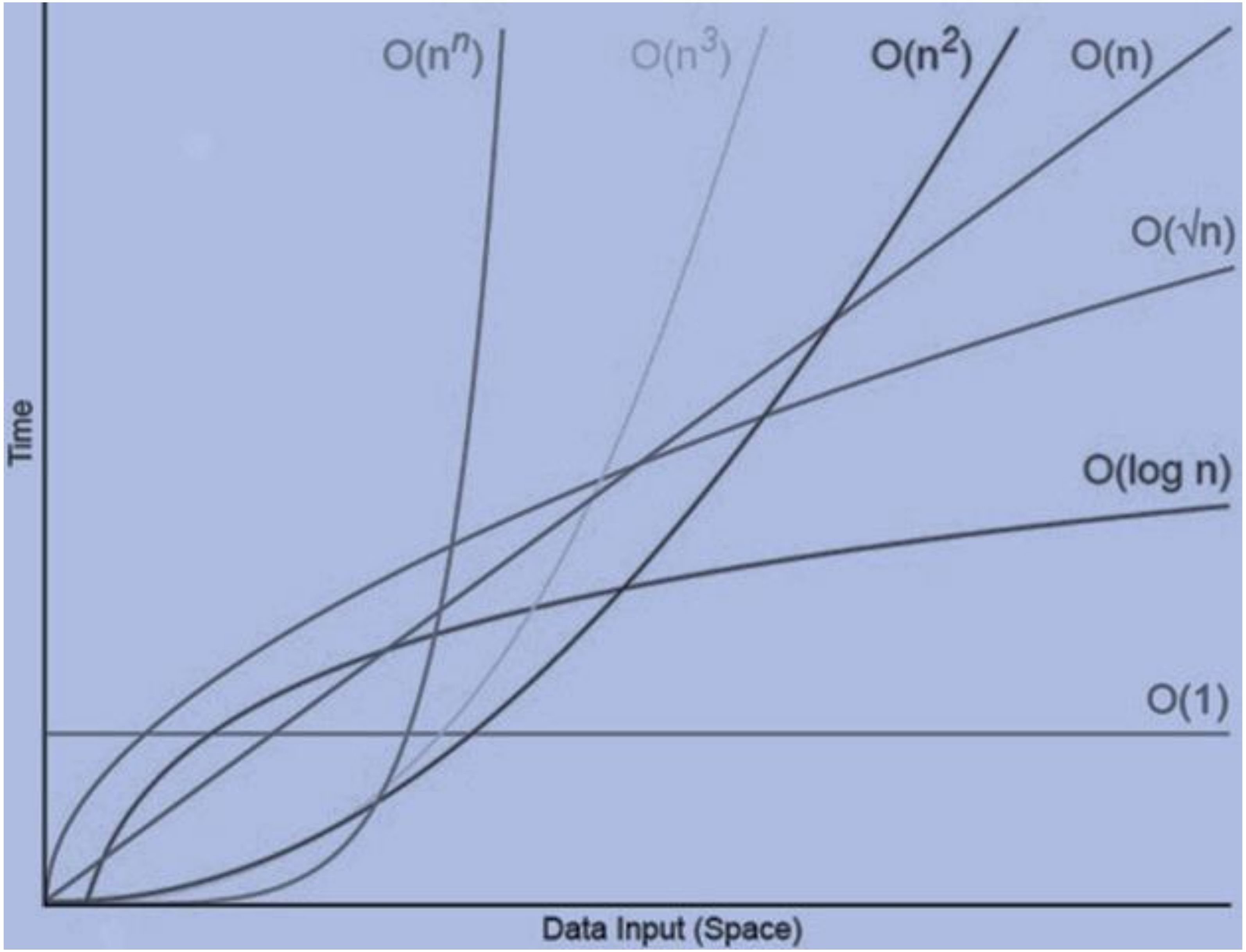
```
# n,k is variable

for i in range(n):
    print(i)
for j in range(m):
    print(j)
```

- ❑ Time complexity of above code is O(n) + O(m) which is O(n+m)

Growth Orders

n	O(1)	O(log(n))	O(n)	O(nlog(n))	O(N^2)	O(2^n)	O(n!)
1	1	0	1	1	1	2	1
8	1	3	8	24	64	256	40xx10^3
30	1	5	30	150	900	10x10^9	210x10^{32}
500	1	9	500	4500	25x10^4	3x10^{150}	1x10^{1134}
1000	1	10	1000	10x10^3	1x10^6	1x10^{301}	4x10^{2567}
16x10^3	1	14	16x10^3	224x10^3	256x10^6	-	-
1x10^5	1	17	1x10^5	17x10^5	10x10^9	-	-

Growth Orders



Growth Orders

Length of Input (N)	Worst Accepted Algorithm
≤10	$O(N!), O(N^6)$
≤15	$O(2^N * N^2)$
≤20	$O(2^N * N)$
≤100	$O(N^4)$
≤400	$O(N^3)$
≤2K	$O(N^2 * logN)$
≤10K	$O(N^2)$
≤1M	$O(N * logN)$
≤100M	$O(N), O(logN), O(1)$

Problems

- Find the complexity of the below program:

```
function(int n)
{
    if (n==1)
        return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Problems

- **Solution:** Time Complexity O(n). Even though the inner loop is bounded by n, but due to break statement it is executing only once.

```
function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        // Inner loop executes only one
        // time due to break statement.
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

- Find the complexity of the below program:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j+n/2<=n; j = j++)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Problems

- **Solution:**

  Time

  $O(n^2 \log n)$

```
void function(int n)
{
    int count = 0;

    // outer loop executes n/2 times
    for (int i=n/2; i<=n; i++)

        // middle loop executes  n/2 times
        for (int j=1; j+n/2<=n; j = j++)

            // inner loop executes logn times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Problems

- ❑ Find the complexity of the below program:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j<=n; j = 2 * j)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Problems

- **Solution:**

  Time

  O(n log²n)

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)

        // Executes O(Log n) times
        for (int j=1; j<=n; j = 2 * j)

            // Executes O(Log n) times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Problems

- Find the complexity of the below program:

```
void function(int n)
{
    int count = 0;
    for (int i=0; i<n; i++)
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

Problems

- **Solution:**

  Time $O(n^5)$

```
void function(int n)
{
    int count = 0;

    // executes n times
    for (int i=0; i<n; i++)

        // executes O(n*n) times.
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                // executes j times = O(n*n) times
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

Contact Me

facebook.com/mloey

mohamedloey@gmail.com

twitter.com/mloey

linkedin.com/in/mloey

mloey@fci.bu.edu.eg

mloey.github.io

THANKS FOR YOUR TIME