PROGRAMMING AND DATA STRUCTURES

ALGORITHM ANALYSIS

HOURIA OUDGHIRI

FALL 2021

OUTLINE

- Measuring the performance of programs
- Algorithm analysis
- Big-O Notation
- Examples

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- Define algorithm design and algorithm analysis
- Determine the time complexity of given algorithms using the Big-O notation
- Compare different solutions to the same problem using algorithm analysis techniques

ALGORITHM ANALYSIS

- Algorithm Design: Finding a computational solution to a problem
- Often many solutions are possible

How to select a solution?

- Use Algorithm Analysis
- Example: Binary Search and Linear Search

ALGORITHM ANALYSIS

- Compare Binary Search/Linear Search
  - Measure the execution time of the two methods
  - Use different sizes of the array
  - Use a random value as the key

ALGORITHM ANALYSIS

```
public static int linearSearch
                (int[] list, int key){
  for (int i=0; i<list.length; i++)
  {
      if (list[i] == key) {
        return i;
      }
  }
  return -1;
}
```

ALGORITHM ANALYSIS

```
public static int binarySearch(int[] list, int key){
   int first, last, middle;
   first = 0;
   last = list.length-1;
   while (first <= last){
      middle = (last + first) / 2;
      if (key == list[middle]) {
         return middle;
      }
      else if (key < list[middle])
         last = middle - 1;
      else
         first = middle + 1;
   }
   return -1;
}
```

ALGORITHM ANALYSIS

```
public static void main(String[] args) {
  final int MAX_SIZE = 100;
  Random rGenerator = new Random();
  int index;
  // Generating random data
int[] data = new int[MAX_SIZE];
for(int i=0; i<data.length; i++)
  data[i] = rGenerator.nextInt(MAX_SIZE);
```

ALGORITHM ANALYSIS

```
 // Searching
int key = rGenerator.nextInt(MAX_SIZE);

int index1 = linearSearch(data, key);

java.util.Arrays.sort(data);
int index2 = binarySearch(data, key);
```

ALGORITHM ANALYSIS

```
// Calculating the execution time
int linearTime, binaryTime;

long linearStartTime = System.nanoTime();
index1 = linearSearch(data, key);
long linearEndTime = System.nanoTime();
linearTime = linearEndTime - linearStartTime;

long binaryStartTime = System.nanoTime();
index2 = binarySearch(data, key);
long binaryEndTime = System.nanoTime();
binaryTime = binaryEndTime - binaryStartTime;
```

Linear vs. Binary Search

Array size	Linear Search Execution time (ns)	Binary Search Execution time (ns)
100	34,388	32,811
1000	46,958	32,893
10,000	166,451	33,831
100,000	1,281,831	35,687
1,000,000	8,443,868	27,282
10,000,000	6,436,336	40,823
100,000,000	27,811,270	40,600

ALGORITHM ANALYSIS

- Comparing the two solutions
  - Implementing the two solutions
  - Comparing the execution times
    - Depend on the machine executing the two programs
    - Depends on the data set

ALGORITHM ANALYSIS

- Algorithm Analysis: Predict the performance of an algorithm before implementing it (coding)
- Determine the upper bound on the performance of the algorithm based on the problem size

ALGORITHM ANALYSIS

```
public static int linearSearch
                (int[] list, int key){
  int iterations = 0;
  for (int i=0; i<list.length; i++){
     iterations++;
      if (list[i] == key) {
        return iterations;
      }
  }
  return iterations;
}
```

ALGORITHM ANALYSIS

```
public static int binarySearch(int[] list, int key){
   int first, last, middle;
   first = 0; int iterations = 0;
   last = list.length-1;
   while (first <= last){
      iterations++;
      middle = (last + first) / 2;
      if (key == list[middle]) {
         return iterations;
      }
      else if (key < list[middle])
         last = middle - 1;
      else
         first = middle + 1;
   }
   return iterations;
}
```

Linear vs. Binary Search

Array size	Linear Search (# iterations)	Binary Search (# iterations)
100	13	6
1000	991	9
10,000	9519	13
100,000	100,000	17
1,000,000	780,668	20
10,000,000	10,000,000	23
100,000,000	99,923,254	27

Big-O notation

- **Growth rate:** How fast an algorithm's execution time (or memory space) increases as the input size increases
- Worst case - Upper Bound
- Guarantees the algorithm execution time can never be higher than the worst case

Big-O notation

- Theoretical approach independent of computers (machines) and specific input
- **Big-O** notation is a mathematical function for measuring algorithm time complexity (or space complexity) based on the input size

Big-O notation

- Time complexity: Execution time as a function of the input size
- Space complexity: Amount of memory space as a function of the input size

ALGORITHM ANALYSIS

```
public static int linearSearch(int[] list,int key){
  for (int i=0; i<list.length; i++){
     if (list[i]==key){
       return i;
     }
  }
  return -1;
}
```

ALGORITHM ANALYSIS

```
public static int linearSearch(int[] list,int key){

  for (int i=0; i<list.length; i++){
      if (list[i]==key){
        return i;
      }
  }
  return -1;
}
```

1 assignment

n comparisons

n incrementations

n comparisons

1 return statement

1 return statement

$$\text{Time}(n) = (1 + 3n + 1) * \text{constant time}$$
$$= (3n + 2) * \text{const}$$

Big-O notation

- Linear Search

$$Time(n) = (3n + 2) \,.\, const = O(n)$$

- Time grows linearly with the input size (`n`)

  `O(n)` - Order of n - Linear Algorithm

- Memory space grows linearly with the input size - size of the array = n - `O(n)`

Big-O notation

- Multiplicative constants and non-dominating terms are ignored
  - **O(100 x n) ≃ O(n/5) ≃ O(n)**
  - **O(n-1) ≃ O(n-1000) ≃ O(n)**
  - **O((5n² + 2n + 11)/7) ≃ O(n²)**
  - **O(1)**: execution time is constant or not related to the input size

Big-O notation

- Useful mathematical formulas:

$1+ 2+ 3+ \ldots + n= n(n+1)/2 - O(n^2)$

$n(n+1)/2= n^2/2+n/2 = n^2+n \cong O(n^2)$

$a^0+a^1+a^2+ \ldots +a^n = a^{(n+1)}-1/a-1 \cong O(a^n)$

$2^0 + 2^1+ \ldots + 2^n = 2^{(n+1)}-1 \cong O(2^n)$

Big-O notation

- What is the order of the following expressions?
  - $(n^2 + 1)^2 / n$
  - $3n^2 + 2n + 1/n$
  - $(n^2 + \log_2 n^2)$
  - $n^3 + 100n^2 + n + 10$
  - $2^n + 25n^3 + 45n$
  - $(n^4 + 2n^2 + 1)/n$

Big-O notation

- Determining Big-O

```
for(int i=0; i<n; i++){

    k = k + 5;
}
```

n times

Constant time

Time Complexity: (3 * n + 1) * const = **O(n)**

Linear Growth

Big-O notation

✦ Determining Big-O

```
for(int i=1; i<= n; i++){
   for(int j=1; j<= n; j++){
      k = k + i + j;
   }
}
```

n times

n times

constant time

Time Complexity: $(1 + 3*n + 3*n^2) * \text{const} = \mathbf{O(n^2)}$

Quadratic Growth

Big-O notation

✦ Determining Big-O

```
for(int i = 1; i <= n; i++){

  for(int j = 1; j <= i; j++){

    k = k + i + j;
  }
}
```

n times

1, 2, … or n times

constant time

Time Complexity: [1 + 3*n + 3*(1+2+…+n)] * const

= n + (n+1)n/2 = **O(n²)** - Quadratic

Big-O notation

Determining Big-O

```
for(int i = 1; i <= n; i++){

    k = k + 4;
}
for(int i = 1; i <=n; i++){

    for(int j=1; j<=20; j++){

        k = k + i + j;
    }
}
```

n times

constant time

n times

20 times

constant time

Time Complexity:

= (1+ 3*n) * const + (1 + 3*n + 3 * 20 * n) * const

= **O(n)**

Big-O notation

✦ Determining Big-O

```
if(list.contains(e)){

   System.out.print(e);
}
else{
 for(Object t: list){

   System.out.print(t);
 }
}
```

n times

constant time

n times

constant time

Time Complexity: (1+ n)*const or ((2* n)+ n)*const = **O(n)**

Big-O notation

✦ Determining Big-O

```
result=1;

for(int i=1; i<=n; i++)

  result = result * a;
```

constant time

n times

constant time

Time complexity: (2 + 3 * n) * const = **O(n)**

Big-O notation

✦ Determining Big-O

```
result = a;

for(int i=1; i<=k; i++)

  result = result * result;
```

constant time

k times, $n = 2^k$

constant time

result = a

Iteration 1: result = a*a = a^2 (2^1)

Iteration 2: result = (a*a)*(a*a) = a^4 (2^2)

Iteration 3: result= (a*a*a*a)*(a*a*a*a)= a^8 (2^3)

Iteration k: result= a*. . . a (2^k)

Time complexity: (2 + 3 * log n) * const = **O(log n)**

Big-O notation

✦ Determining Big-O

```
int count = 1;

while(count < n)

  count = count * 2;
```

constant time

n/2 times

constant time

Time complexity: (1 + 2 * n/2) * const = **O(n)**

Big-O notation

✦ Analyzing Binary Search

```
public static int binarySearch(int[] list, int key) {
   int first, last, middle;
   first = 0;
   last = list.length-1;
   while (first <= last){
     middle = (last + first) / 2;
     if (key == list[middle]) {
       return index;
     }
     else if (key < list[middle])
       last = middle - 1;
     else
       first = middle + 1;
    }
    return -1;
}
```

Big-O notation

✦ Analyzing Binary Search

Iteration 1: **n/2**

Iteration 2: **n/4**

Iteration k = **$n/(2^k)$**

Last iteration x: $n/2^x = 1$ (one element left)

$x = \log_2(n)$

Binary Search: O(log n) - Logarithmic growth

Analyzing Selection Sort

```
public static int selectionSort(int[] list) {
  for(int i=0; i<list.length-1; i++){
    // find the smallest element
    int currentMin = list[i];
    int currentMinIndex = i;
    for(int j=i+1; j<list.length; j++){
     if (list[i] < currentMin){
        currentMin = list[i];
        currentMinIndex = i;
      }
    }
    if(currentMinIndex != i){
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
  }
}
```

Big-O notation

✦ Analyzing Selection Sort

```
Iteration1(outer loop) i=0
    (n-1)iterations(inner loop) 1, n

Iteration 2 (outer loop) i=1
    (n-2) iterations (inner loop) 2, n

Iteration k (outer loop) i=k-1
    (n-k) iterations (inner loop) k, n

Iteration n-1 (outer loop) i=n-2
    1 iteration (inner loop) n-1, n-1

(n-1) + (n-2) + … + 1 = (n-1)n/2
```

Selection Sort: $O(n^2)$ - Quadratic growth

Big-O notation

- Analyzing Recursive Fibonacci sequence

```
public static long fibonacci(long n) {

    if(n == 1 || n == 2)
        return 1;

    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Big-O notation

✦ Analyzing Recursive Fibonacci sequence

$$Time(n) = Time(n-1) + Time(n-2)$$

$$Time(n) \sim 2 * Time(n-1)$$

$$Time(n-1) = 2 * Time(n-2)$$

$$Time(n) = 2 * 2 * Time(n-2)$$

…

$$Time(n) = 2^k * Time(n-k)$$

$$Time(n) = 2^{(n-2)}\ Time(2)$$

$$Time(n) = 2^{(n)} * constant$$

Recursive Fibonacci: $O(2^n)$ -

Exponential growth

Big-O notation

- Analyzing Iterative Fibonacci sequence

```
public static long fibonacci(long n) {
   long f0=0,f1=1, f2=1;
   if(n == 1 || n == 2)
         return f1;
   for(int i=3; i<=n; i++){
         f0=1;
         f1=f2;
         f2=f0+f1;
   }
   return f2;
}
```

Big-O notation

- Analyzing Iterative Fibonacci sequence

```
Time Complexity: (8 + 5 * (n-3)) * const

Iterative Fibonacci: O(n) - Linear growth
```

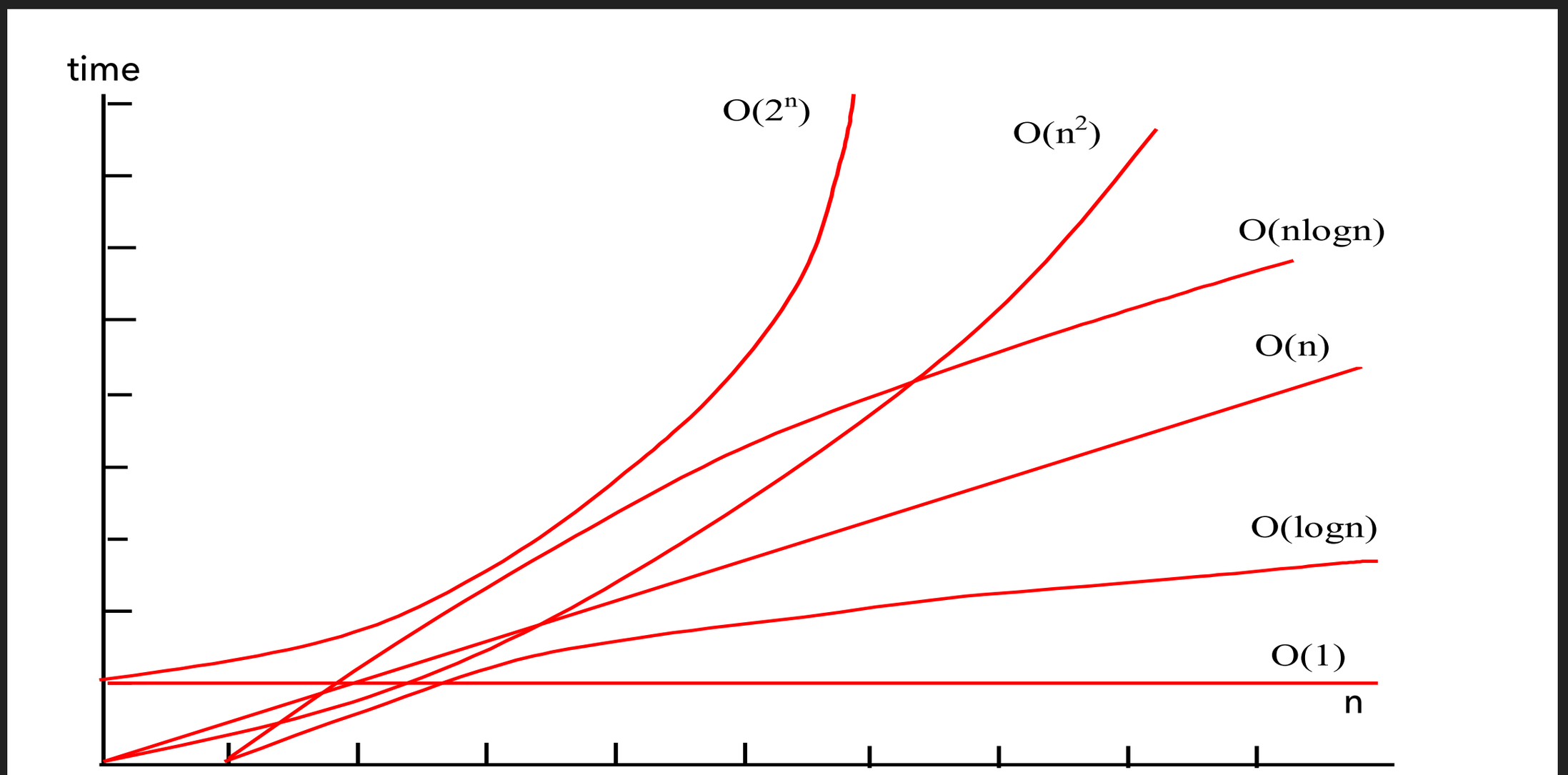
Big-O notation

Common growth functions

Growth function type	Big-O notation
Constant time	`O(1)`
Logarithmic time	`O(log n)`
Linear time	`O(n)`
Log-Linear time	`O(n logn)`
Quadratic time	$O(n^2)$
Cubic time	$O(n^3)$
Exponential time	$O(2^n)$
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$	

Big-O notation

Common growth functions



Big-O notation

- Put the following growth functions in order
  - $\frac{5n^3 + 3n^2 + n}{1000}$
  - $44logn + 22n$
  - $10.nlogn - 2n$
  - 500
  - $\frac{2^n + 3n^5}{45}$
  - $2n^2 + n + 50$

Big-O notation

- Determine the time complexity of the code below using the Big-O notation

```
for (int i = n/2; i > 0; --i){
    int x = n * 3;
    while(x > 1){
      for (int j=0; j < 100; j+=2)
        System.out.prinltn(“X: “ + x);
      x = x / 2;
    }
}
```

Efficient Algorithms

- Designing algorithms while minimizing their time complexity
- Binary search improves linear search complexity by having the data sorted prior to the search
- Iterative Fibonacci sequence reuses prior calculations to determine next terms

Efficient Algorithms

Finding the **GCD** (Greatest Common Divisor) of two integers **m** and **n**

- Finding the largest divisor of two numbers starting from 1 up
- Finding the largest divisor of two numbers starting from n down
- Finding the largest divisor less than or equal to n/2 - Divisor of n cannot be greater than n/2
- Euclid's GCD Algorithm

Efficient Algorithms

GCD Algorithm - V1

```
public static int gcd(int m, int n){
   int divisor = 1;
   for(int i=2; i<m && i<n; i++){
      if(m%i == 0 && n%i == 0)
         divisor = i;
   }
   return divisor;
}
```

Time complexity: (2+ 7 * (n-2)) * const = **O(n)**

Efficient Algorithms

GCD Algorithm - V2

```
public static int gcd(int m, int n){
    int divisor = 1;
    for(int i = n; i >= 1; i--){
      if(m%i == 0 && n%i == 0){
          divisor = i;
          break;
      }
    }
    return divisor;
}
```

Time complexity = (2 + 5 * n + 2) * const = **O(n)**

Efficient Algorithms

GCD Algorithm - V3

```
public static int gcd(int m, int n) {
   int divisor = 1;
   if(m%n == 0) return n;
   for(int i = n/2; i >= 1; i--)
      if(m%i == 0 && n%i == 0){
         divisor=i;
         break;
      }
   return divisor;
}
```

Time complexity = 3 * const + 5 * const * n/2 + 2 = O(n)

Efficient Algorithms

GCD Algorithm - V4 (Euclid's)

```
public static int gcd(int m, int n){
   if(m%n == 0)
      return n;
   else
      return gcd(n, m%n);
}
```

Time complexity: (log n) recursive calls at most = **O(log n)**

Efficient Algorithms

Finding the GCD of two integers

- Version 1 - Worst case and always **O(n)**
- Version 2 - Worst case : **O(n)** - runs faster than version 1 on average
- Version 3 - Worst case: **O(n)** - runs twice faster than version 2
- Version 4 - Worst case **O(log n)**

Summary

- Algorithm analysis - predict the performance of an algorithm independently of its implementation
- Estimate the time complexity or space complexity (growth rate as a function of the problem size)
- Big-O notation - Upper bound for growth rate
- Algorithm efficiency - reduce time complexity - tradeoff between time and space complexity