Time and Space complexity

Agenda

- Overview and notations
- Algorithm complexity
- Exercises

Overview and notations

Overview

- Computational complexity theory aims to categorize different algorithmic problems according to their difficulty

- Uses mathematical models to provide estimates in terms of resources needed to solve a particular problem such as:

  - **time**: the predicted execution time

  - **space**: the predicted memory required by the problem

Overview

- In practice complexity theory can be used to statically determine what is the required execution time or memory for a piece of code

- Applies for pretty much any programming language and program logic (not only complex algorithms)

Overview

- To provide those estimates complexity theory uses several notations used to predict:
  - worst case
  - best case
  - average case

Complexity

- Complexity is expressed as a function of the inputs

- For an algorithm depending on input **n**, complexity might be expressed a function **f(n)** of that input

```
n -> f(n)
```

- For multiple inputs complexity might a function of multiple parameters

```
n, m -> f(n, m)
```

Notations

- Theta notation measures average case complexity
- It is measured by a function f(n) such that there are two constants c1 and c2, which for sufficiently large n (n0) can bound f(n) by g(n)

$$\Theta(g(n)) = \{ f(n) \mid \exists c1,\ c2 > 0,\ \exists n0 : \forall n \geq n0,\ 0 \leq c1.g(n) \leq f(n) \leq c2.g(n) \}$$

Notations

- O notation (big or small) measures worst case complexity

- Big-O is measured by a function f(n) such that there is a constant c which for sufficiently large n (n0) can upperbound f(n) by g(n)

$$O(g(n)) = \{ f(n) \mid \exists c > 0, \exists n0 : \forall n \geq n0, \; 0 \leq f(n) \leq c.g(n) \}$$

- For small-o there is any constant c > 0 for sufficiently large n(n0)

$$o(g(n)) = \{ f(n) \mid \forall c > 0, \exists n0 : \forall n \geq n0, \; 0 \leq f(n) < c.g(n)$$

Notations

- Omega notation (big or small) measures best case complexity

- Big-omega is measured by a function f(n) such that there is a constant c which for sufficiently large n (n0) can lowerbound f(n) by g(n)

```
Ω(g(n)) = { f(n) | ∃c > 0, ∃n0 : ∀n ≥ n0,
                   0 ≤ c.g(n) ≤ f(n)
```

- For small-omega there is any constant c > 0 for sufficiently large n(n0)

```
ω(g(n)) = { f(n) | ∀c > 0, ∃n0 : ∀n ≥ n0,
                   0 ≤ c.g(n) < f(n)
```

Basic terminology

$O(1)$	constant
$O(\log(n))$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential
$O(n!)$	factorial

Big-O notation is most widely used in practice

Amortized complexity

- Complexity might not be very precise in many cases as it depends on the operations and not on the algorithm itself

- That is why amortized analysis tries to provide a more accurate measure for time and space complexity

- The general idea is that we first determine the total cost of a sequence operations and divide the results by the number of these operations

```
T(n)/n
```

Amortized complexity

- Example (amortized complexity is O(n)):

```
public double f(double[] prices,
              double[] discounts) {
       double result = 0;
       for(int i = 0; i < n; i++) {
              result+= prices[i];
              // if there are 10 items or more
              // then discounts apply
              if(i == 10) {
                     for(int j = 0; j < n; j++) {
                            result -= discounts[j];
                     }
              }
       }
       return result;
}
```

Algorithm complexity

Algorithm complexity

- We will define how to weight each type of construct in the Java programming language in order to measure properly time complexity:
  - simple operations like variable assignment, expressions, if/swtich statements etc. are considered constant
  - loops (for/while) take linear time if number of iterations is linear on an input parameter
  - recursive calls add another method invocation, hence multiply to the complexity of the method (without the recursive calls)

Constant complexity

Example:

```
public int f(int x, int y) {
       return x + y;
}
```

Linear complexity

Example:

```
public int f(int n, int[] numbers) {
        int result = 0;
        for(int number : numbers) {
                result += number;
        }
        return result;
}
```

Quadratic complexity

Example:

```
public int f(int n, int[] numbers) {
        int result = 0;
        for(int n1 : numbers) {
                for(int n2 : numbers) {
                        result += n1 + n2;
                }
        }
        return result;
}
```

Polynomial complexity

Example:

```
public int f(int n, int[] numbers) {
        int result = 0;
        for(int n1 : numbers) {
                …
                for(int n10 : numbers) {
                        result += n1 + … + n10;
                }
        }
        return result;
}
```

Logarithmic complexity

Example:

```
int indexedBinarySearch(List<? extends Comparable<? super T>>
                list, T key) {
        int low = 0;
        int high = list.size()-1;
        while (low <= high) {
                int mid = (low + high) >>> 1;
                Comparable<? super T> midVal = list.get(mid);
                int cmp = midVal.compareTo(key);
                if (cmp < 0)   low = mid + 1;
                else if (cmp > 0) high = mid - 1;
                else return mid; // key found
        }
        return -(low + 1);  // key not found
}
```

Above method is called from **java.util.Collections#binarySearch**

Exponential complexity

Example:

```
public int f(int n, int c) {
        int result = 0;
        for(int i = 2; i <= n; i++) {
                result += Math.log(i);
        }

        if(c > 1) {
                result += f(n, c - 1);
        }
        return result;
}
```

Factorial complexity

Example:

```
public int f(int n) {
	int result = 0;
	for(int i = 2; i <= n; i++) {
		result += Math.log(i);
	}

	if(n > 1) {
		result += f(n - 1);
	}

	return result;
}
```

Exercises

What is the complexity ?

```
public double f(int n) {
	double a = 0;
	for(int i = 0; i < n; i++) {
		for(int j = i; j < n; j++) {
			for (int k = n + i + j - 3; k < n; k++) {
				a = a + Math.log(k);
			}
		}
	}
	return a;
}
```

What is the complexity ?

```
public double f(int n) {
        double a = 0;
        for(int i = 0; i < n; i++) {
                for(int j = i; j < n; j++) {
                        for (int k = n + i + j - 3; k < n; k++) {
                                a = a + Math.log(k);
                        }
                }
        }
        return a;
}
```

Answer: O(n^2)

What is the complexity ?

```
public double f(int n, int[] numbers) {
        double a = 0;
        for(int i = 0; i < n - 4; i++) {
                for(int j = i; j < i + 4; j++) {
                        for (int k = i; k < j; k++) {
                                a = a + numbers[i];
                        }
                }
        }
        return a;
}
```

What is the complexity ?

```
public double f(int n, int[] numbers) {
        double a = 0;
        for(int i = 0; i < n - 4; i++) {
                for(int j = i; j < i + 4; j++) {
                        for (int k = i; k < j; k++) {
                                a = a + numbers[i];
                        }
                }
        }
        return a;
}
```

Answer: O(n)

What is the complexity ?

```
public int f(int n) {
        int result = 0;
        for(int i = 1; i <= n; i++) {
                result += Math.cos(i);
        }

        if(n > 1) {
                result += f(n/2);
        }

        return result;
}
```

What is the complexity ?

```
public int f(int n) {
       int result = 0;
       for(int i = 1; i <= n; i++) {
              result += Math.cos(i);
       }

       if(n > 1) {
              result += f(n/2);
       }

       return result;
}
```

Answer: O(n log(n))

What is the complexity ?

```
// mergeSort(n,  0,  n.length - 1);
public void mergeSort(Comparable [ ] a,
       int left, int right) {
       if( left < right )
        {
              int center = (left + right) / 2;
              mergeSort(a, left, center);
              mergeSort(a, center + 1, right);
              // merge is O(n)
              merge(a, tmp, left, center + 1, right);
       }
}
```

What is the complexity ?

```
// mergeSort(n,  0,  n.length - 1);
public void mergeSort(Comparable [ ] a,
       int left, int right) {
       if( left < right )
        {
              int center = (left + right) / 2;
              mergeSort(a, left, center);
              mergeSort(a, center + 1, right);
              // merge is O(n)
              merge(a, tmp, left, center + 1, right);
       }
}
```

Answer: O(n log(n))

What is the complexity ?

```
void bubbleSort(int arr[], int n)
{
for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]) {
                    swap(arr[j], arr[j+1]);
              }
}
```

What is the complexity ?

```
void bubbleSort(int arr[], int n)
{
for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]) {
                    swap(arr[j], arr[j+1]);
              }
}
```

Answer: O(n^2)

What is the complexity ?

```
public double f(int n, int[] numbers) {
        double a = 0;
        for(int i = n; i > 0; i/=2) {
                for(int j = i; j < n; j*=2) {
                        for (int k = 0; k < n; k+=2) {
                                a = a + numbers[i];
                        }
                }
        }
        return a;
}
```

What is the complexity ?

```
public double f(int n, int[] numbers) {
        double a = 0;
        for(int i = n; i > 0; i/=2) {
                for(int j = i; j < n; j*=2) {
                        for (int k = 0; k < n; k+=2) {
                                a = a + numbers[i];
                        }
                }
        }
        return a;
}
```

Answer: O(n log(n)^2)

Questions ?